

Master Thesis

Development and Realization of
a distributed control algorithm for an inverted rotary pendulum
using FreeRTOS with TTCAN communication

Submitted by:

Bui, Hoang Dung

Matr.-Nr.: 976477

Supervised by:

Prof. Dr. Roman Obermaisser

Dr.-Ing. Walter Lang

Master Thesis's Report

Development and Realization of
a distributed control algorithm for an inverted rotary pendulum
using FreeRTOS with TTCAN communication

Table Content

1 Introduction.....	1
1.1 Overview	1
1.2 Master thesis objectives.....	2
1.3 Report's structure	3
2 Basic concepts and Related Works.....	5
2.1 Related Work.....	5
2.1.1 The mechanical design	5
2.1.2 The results from the student work.....	7
2.2 Embedded C	9
2.2.1 Basic concepts in Embedded C.....	10
2.2.2 Control Statements	11
2.2.3 Pointer, Arrays and Structures	13
2.3 FreeRTOS	14
2.3.1 Introduction.....	14
2.3.2 Used FreeRTOS Library Classes	18
2.4 CAN and Time Trigger CAN.....	20
2.4.1 Controller Area Network – CAN	21
2.4.2 Time Trigger CAN – TTCAN	22
2.5 AT90CAN128 Microcontroller	26
2.5.1 The specific knowledge about AT90CAN128.....	26
2.5.2 CAN Controller in AT90CAN128	28
3 Modeling and Microcontroller's Structures	33
3.1 The nonlinear IRP Model	33
3.2 Microcontroller Structures.....	37
3.2.1 TTCAN Scheduler	38
3.2.2 Time Trigger Matrices.....	40
3.2.3 TTCAN_Scheduler Task and CAN ISR	43
3.2.4 Microcontroller Tasks.....	45
4 Implementation.....	52
4.1 Nonlinear IRP model.....	52
4.2 The common parts	54

4.2.1 CAN Tasks	54
4.2.2 CAN ISR	59
4.3 SEN-uC	61
4.3.1 Specific External Interrupt Service Routines	61
4.3.2 Data_Preparation() Task.....	62
4.4 LCD-uC	63
• Data_Reception() Task	63
• LCD_Display() Task.....	64
4.5 PUSB-uC.....	65
• PWM_Calculation() Task	65
• PushedButton_Detection() Task.....	67
4.6 MOTO-uC.....	68
4.6.1 Timer/Counter 3 Compare Match Interrupts	69
4.6.2 DC-motor Control Tasks	70
4.7 Parameters Estimation	74
4.7.1 The IRP parameters	74
4.7.2 LQR parameters.....	76
4.7.3 PWM and Sampling frequencies	77
5 Results and Discussion	78
5.1 Results	78
5.1.1 The nonlinear IRP model behavior	78
5.1.2 The IRP machine Operation.....	80
5.2 Discussion	86
5.2.1 Sampling and PWM frequency.....	86
5.2.2 Mechanical Structures.....	87
6 Conclusion	88
Reference	89
Bibliography.....	89
Appendix.....	90

1 Introduction

The Inverted Rotary Pendulum - IRP is a machine which is invented in 1992 by Katsuhisa Furuta. It contains a driven arm and a pendulum which is attached on the arm. Both the arm and pendulum can rotate freely in the horizontal and vertical plane respectively. In order to control the IRP, it is needed the cooperation of several different fields such as mechanics, control theory and microcontroller science. From the mechanical field, the IRP is a machine which transmits the energy by an input – the DC-motor to control two rotations. In the control system theory's view, the IRP is nonlinear system and only two states can be measured directly. From the microcontroller science side, microcontrollers should be fast enough to produce appropriate actions to response any variation of the IRP. The IRP is considered to work well if its pendulum can stays at upright position forever. Therefore this project's goal is step by step to find a solution which makes the IRP operating well in the specific requirements. In this chapter would give an overview idea about the real IRP machine, the master thesis goal in detail and the report's structure.

1.1 Overview

This master thesis project is the step to go further from the student work *"Modeling and Simulation of an Inverted Rotary Pendulum"* [Bui, 2013]. In the student work, the approximate linear model of the IRP had built. Moreover, the control theory which controls IRP had also established, it was State Space method. That method was applied to the Simulink model then the model's response was stable and worked at upright position. However, there are differences between the model and the machine. That is the built model was linear model whereas the machine contains nonlinear parts. That issue would be solved in this thesis. Moreover, this project would go deeper to the control problem of the machine. The IRP machine is shown on the figure 1.1

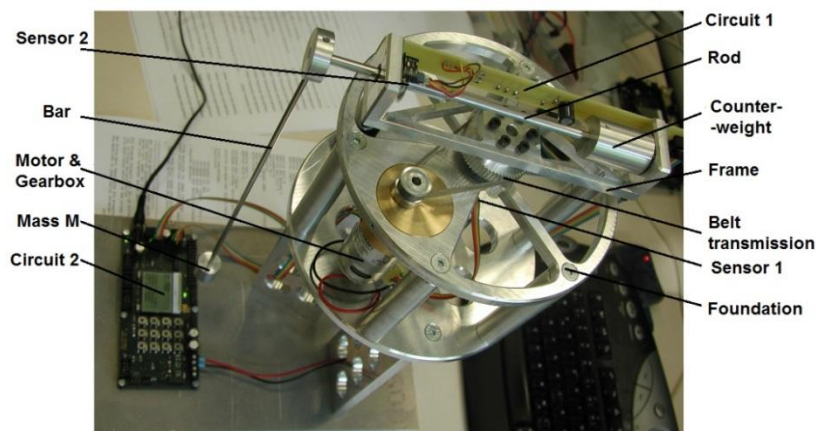


Figure 1.1 the IRP Overview

On the figure 1.1, the main parts of the IRP can be seen: the pendulum/mass M, the gearbox and DC-motor, belt transmission, the Frame, tow sensor and the foundation. The IRP movement begins from the DC-motor. The movement is transferred to the Frame via the gearbox and belt transmission. When the Frame rotates, it makes the mass/pendulum swinging up to the upright position. The two movements which are measured by two encoders are the angle positions of the Frame and the mass M. The two measured angles are called Arm angle and Pendulum angle respectively.

Signals from both encoders which are primitive processed by circuit 1 are sent to the integrated circuit 2. The integrated circuit 2 is a compound circuit board which consists of four microcontrollers AT90CAN128 and several equipment such as a LCD DOGS102-6, pushed button matrix, LEDs, signal ports, fours PCA82C250, L6205D. The four microcontrollers process the data which are used to control DC-motor. The pushed button matrix and LCD DOGS102 are used to communicate with users (Human Machine Interface – HMI). The LCD shows the information of the IRP system. The button matrix receives the commands from users then manipulates the state of the IRP system. Four PCA82C250s are the bridge between the CAN controller and the CAN Bus. The L6205D receives the signal PWM from I/O port of microcontroller and add more energy to control DC-motor directly.

There are four microcontrollers used to control the IRP. The first microcontroller detects the movements of Arm angle and Pendulum angle and transmits the new angles to other microcontroller. The second microcontroller displays all the IRP information on the LCD. The third one receives signal from the button matrix and control the IRP working or stopping. The last microcontroller controls the rotation of DC motor, thereby control the IRP movement.

1.2 Master thesis objectives

As mentioned earlier, this thesis is completed the work-in-progress from the student work. The basic and critical duty is to write the codes for four microcontrollers to control the IRP machine working at upright position and is stable there. In order to implement that duty, some minimum objectives should be fulfilled. They are:

- Build nonlinear model of IRP.
- Define constraint times which can keep the system working stable.
- Handle with the sensor signal and the backlash.
- Check capacity of Microcontrollers to meet the IRP control requirements.

- Write the program to control the IRP working at upright position with the stability time about 5 seconds.

As the basic objectives are completed, the master thesis goals are extended. There are some extra parts which would be added to the system to make it friendly to human. The additional objectives are:

- Establish two separate programs to control IRP in two phases: Swing up phase and Upright phase.
- Human – Machine Interface (HMI): show IRP information on LCD, and some buttons to control IRP
- Be able to change the reference input of arm angle.
- Be able to transmit the signal of IRP angles to computer, and shown it graphically.

That is the objectives of this master thesis project. Subsequently, the report structure would be introduced.

1.3 Report's structure

The report included six chapters. Chapter sequence was formed to assist reader in understanding step by step how the student's work was implemented. Knowledge was presented in appropriate chapters and described how they can be applied to this specific IRP.

Chapter 1 presents general knowledge about IRP, objectives and report's structure.

Chapter 2 describes the related works and basic concepts which would be used to control the IRP machine.

Chapter 3 introduces the designed work on each microcontroller. It starts with the general structures in the four microcontroller such as the Scheduler, the CAN message, TTCAN_Scheduler task and CAN ISR. Then it works with the specific tasks which are for the specific microcontrollers.

Chapter 4 would implement the design in the programming language. In this chapter, the description of the microcontroller structures are converted to the Embedded C, FreeRTOS code and they will actually control the IRP machine. It also describes the parameter estimation process.

Chapter 5 would present the results from chapter 4. The response in the nonlinear model and the IRP machine will be shown and discussed. The learned knowledge is also discussed here.

Chapter 6 concludes about the master thesis project. The achieved goals were reached in this work.

In the end of the work, the declaration, reference and appendix – the program codes in detail of the former chapters are presented.

The aforesaid information plays as a key role to assist readers in having an overview of this master thesis. The second chapter hereafter presents some basic concepts and related works regarding the IRP.

2 Basic concepts and Related Works

This chapter presents the knowledge which is the foundation to develop the IRP operation. The first section of chapter would recall the related works, which are already completed and their results will be applied in this master thesis. The second section presents the basic concepts which are deployed to program the IRP control code such as Embedded C, FreeRTOS, TTCAN and AT90CAN128 microcontroller. Because of the enormous knowledge of those fields, only the specific knowledge will be introduced.

2.1 Related Work

In order to carry out the controlled works of IRP machine in upright position, that is obviously essential to have a foundation. The first foundation is the existence of a real IRP machine, which is outlined the first chapter. The technical parameters, the engineering drawings would be described in detail in the section 2.1.1. The second foundation is the control theory, which is responsible to keep the mass of IRP machine at upright position. That theory was established in my student work [Bui, 2013], and was applied in the IRP model. The response of that model was satisfied the control requirements in the response time and the stable. The key points of both foundations would be the following sections.

2.1.1 The mechanical design

This section presents the engineering draws of IRP machine's structure. The overview of the IRP in design phase is shown in the figure below.

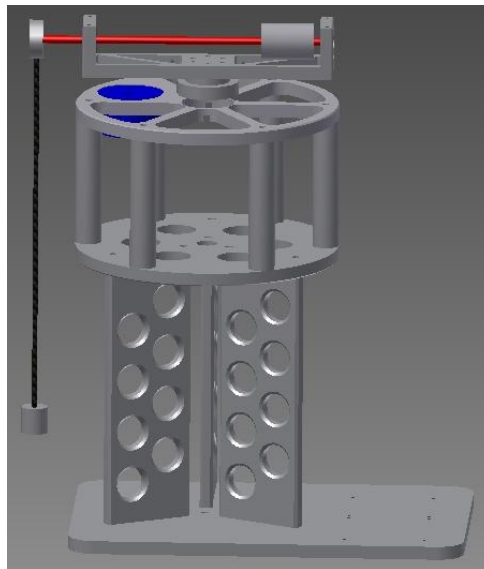


Figure 2.1: The 3-D draw of the IRP

In this draws the blue item represent for the DC-motor. The belt transmission and the bearing are missing. From the 3-D view, it is projected in the front and top direction. The results are the top view and front view draws with more detail about the IRP. From the front view of the IRP, its height is 400 mm, the length is 350 mm and the width is 200 mm.

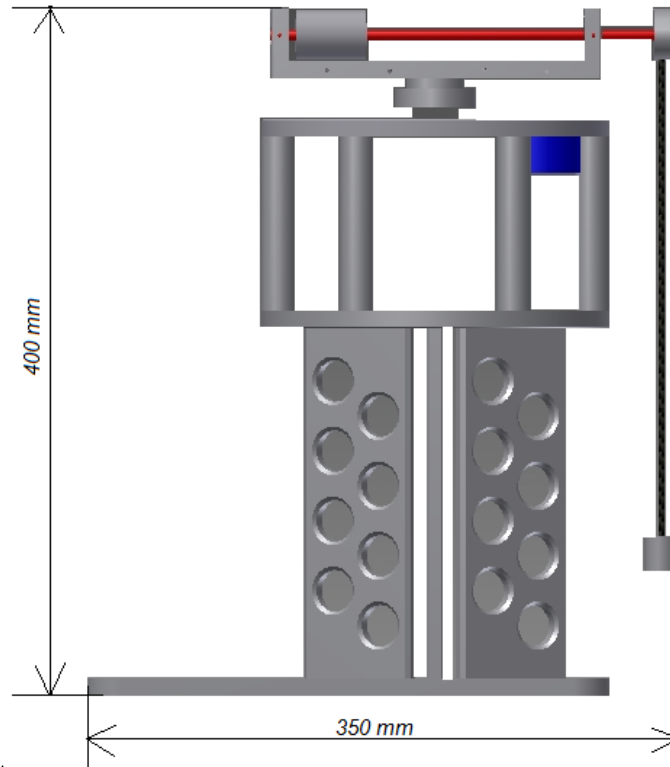


Figure 2.2 Front view of the IRP

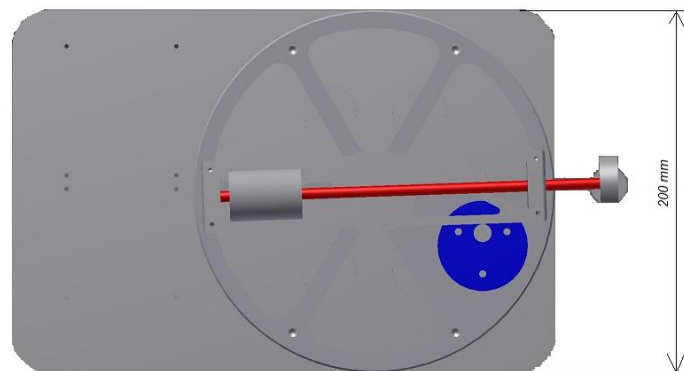


Figure 2.3 Top view of the IRP

The front view and the top view give the image of dimension of the IRP, the distance among components. It also gives a visual view of the mechanical structure about the machine. Based on the dimension, the procurement of the components such as DC-motor, gearbox, belt transmission can perform to meet the requirements. The engineering draws then are sent to the mechanical shop, which is charge of the production and assembly the foundation and other components.

2.1.2 The results from the student work

The works which were accomplished in the student work would be recalled in this section. In the student work, the linear IRP model was built. Moreover, the adequate control theory was also found that was State Space Control method. In this method, the system equations were set to become the equation system of the first order. The system states and inputs were set to be respective vectors. The key point of state space method is in order to keep the system in steady state, the system states must go to zero as the time run to infinity. It is obvious that the opened IRP was impossible to hold itself at the upright position. To hold the IRP at the desired position, a feedback part which was also called *the controller* was added to the IRP. This part did nothing else than measure the output, multiplied with the *feedback parameters* then transmitted the result to compare with the reference input. If there is a difference between the reference input and the output, the difference will be converted to voltage signal and control the DC-motor to run to the desired position. If the difference is zero, the IRP do nothing. The listed figures below shows the Simulink model of the IRP.

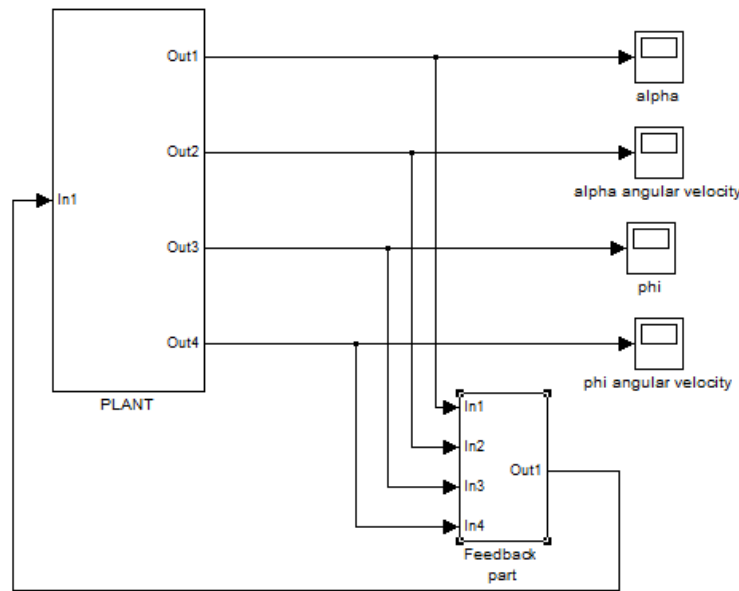


Figure 2.4 Plant with feedback signal

In the figure above describes the IRP model without the reference input or can say the reference input is zero. There are four states in the system. The states are collected by the Feedback component then the feedback signal is sent back to the PLANT to adjust the DC-motor movement. The structure of the PLANT and Feedback component is shown on the figure below.

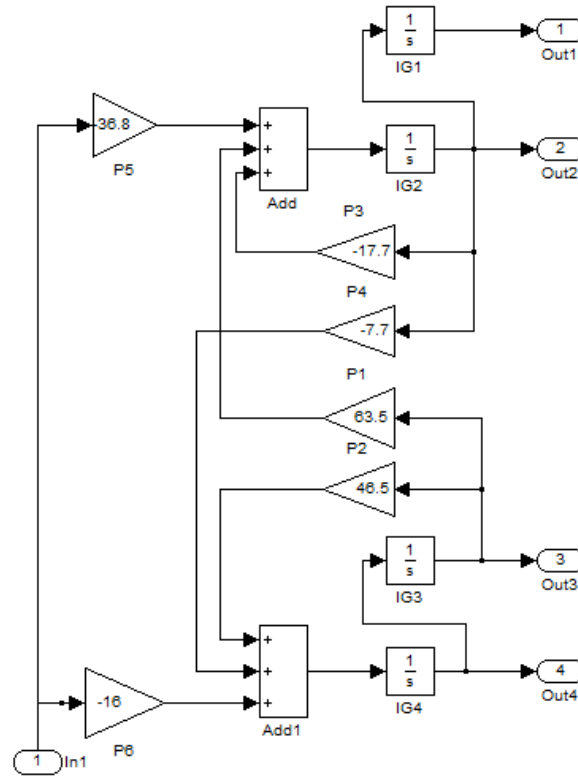


Figure 2.5: Plant Part

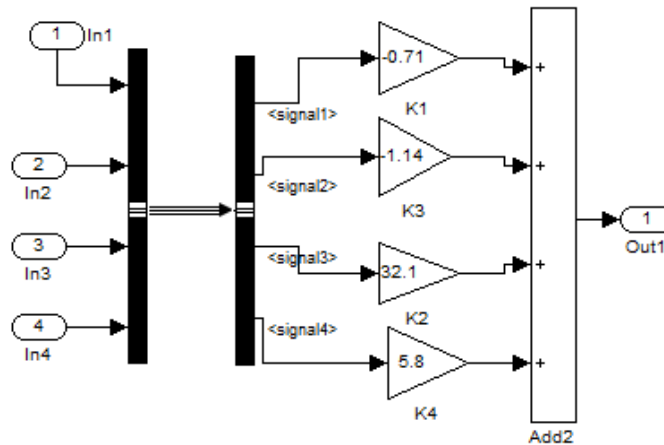


Figure 2.6: Feedback component

As explained, the feedback part is the key part to keep IRP working properly. In other work, the feedback parameters are the critical things of the system. There are some ways to calculate those parameters. However, in the student work the parameter was calculated by linear quadratic regulator (LQR) method, which was able to balance between the system effort and the output response. Specifically, the parameters were processed by function *lqr()* in the Matlab software.

The formula to calculate the feedback signal is:

$$\text{Feedback} = K1 * \text{Arm_Position} + K2 * \text{Pendulum_Position} + K3 * \text{Arm_Velovity} + K4 * \text{Pendulum_Velocity}$$

And the controlled signal is equal to Feedback signal but with opposite sign:

$$\text{Controlled_Signal} = -\text{Feedback}$$

In order to extend the IRP capability, an integral component was added beside the feedback part. With this additional part, the IRP is able to work at arbitrary position of the arm (in working range) and the mass is still hold at the upright position.

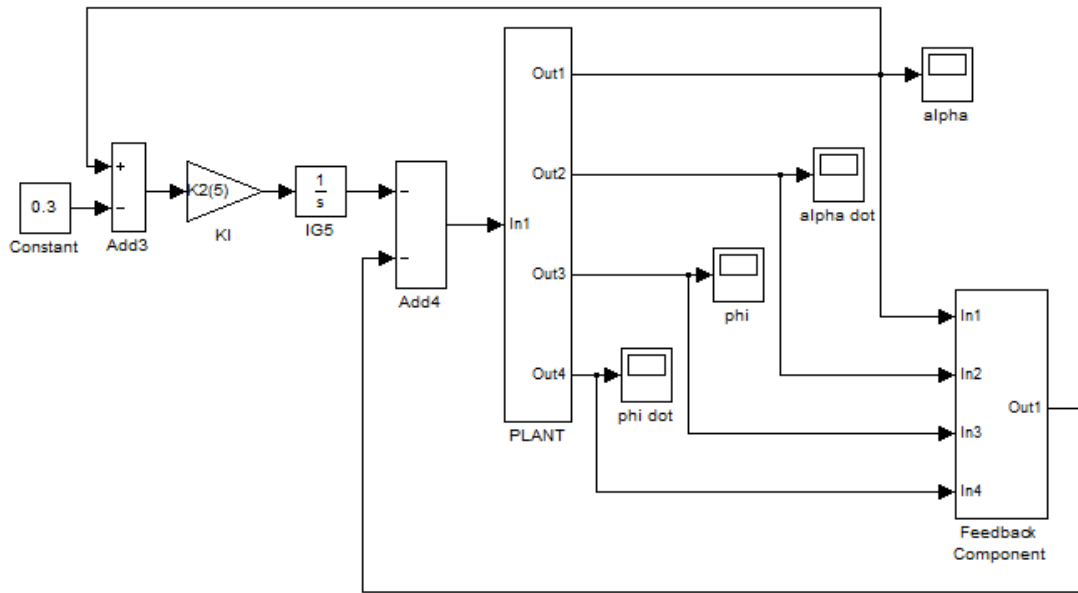


Figure 2.7: The full IRP model

The new component produces an added signal by formula:

$$\text{Added_Signal} = (\text{Arm_Position} - \text{Constant}) * dt * Ki$$

Where dt is the time period between two times taken the sensor signal which is equal to 3 milliseconds.

In this content, the controlled signal is different to the feedback signal, and the controlled signal is calculated by the formula:

$$\text{Controlled_Signal} = -\text{Added_Signal} - \text{Feedback}$$

In addition, the moment of inertia of IRP was calculated, and would be used in this master thesis work.

2.2 Embedded C

Embedded C is branch of C programming language which is developed to apply to embedded microcontroller application. The main difference between the C language and Embedded C is what are they designed for. The first language is designated for personal computers which have an operating system. When a program was

executed on the computer, it returns to the operating system. Meanwhile, the second one is designed to work on microcontrollers which have no operating system thereby it is not allowed fall out of the program any time. Hence every embedded system microcontroller applications have an infinite loop into it [Barnett et.al, 2003].

In an embedded C program, functions are formed by sets of basic instructions and treated as higher-level operations, which are then combined to form a program. The program begins from the *main()* function, the one is set to be the highest priority task in Embedded C. The *main()* then invokes other functions to implement their duties. In many case, *main()* will contains only a few statements that do nothing more than initialize and steer the operation of program from one function to another.

In this section, the knowledge of Embedded C which involve to this project will be introduced.

2.2.1 Basic concepts in Embedded C

In this part, it is started by introducing the symbol conventions in Embedded C. The next one is kinds of variables, constant, I/O operation and bitwise operator.

Just as C language, there are some symbol conventions to compile the program in Embedded C. They are listed in the table below:

Symbol	Meaning
;	A semicolon is used to indicate the end of an expression. An expression in its simplest form is a semicolon alone.
{ }	Braces “{}” are used to delineate the beginning and the end of the function’s contents. Braces are also used to indicate when a series of statements is to be treated as a single block.
“text”	Double quotes are used to mark the beginning and the end of a text string.
// or /* ... */	Slash-slash or slash-star/star-slash are used as comment delimiters.
#include	Add external library to the program

Because of the memory restriction in microcontrollers, the memory should be used in efficient way. Therefore, Embedded C’s creator built up kinds of variables which is used to optimize in memory.

Type	Size (Bits)	Range
bit	1	0, 1
char	8	–128 to 127
unsigned char	8	0 to 255
signed char	8	–128 to 127
int	16	–32768 to 32767
short int	16	–32768 to 32767
unsigned int	16	0 to 65535

signed int	16	–32768 to 32767
long int	32	–2147483648 to 2147483647
unsigned long int	32	0 to 4294967295
signed long int	32	–2147483648 to 2147483647
float	32	±1.175e-38 to ±3.402e38
double	32	±1.175e-38 to ±3.402e38

CONSTANTS: are fixed values – they are not allowed to be modified as the program executes. Constant can be described by some kind of numeric form:

- Decimal form without a prefix (such as 1234)
- Binary form with **0b** prefix (such as 0b101001)
- Hexadecimal form with **0x** prefix (such as 0xff)
- Octal form with **0** prefix (such as 0777)

TYPE CASTING: The cast, called out in parentheses, convert the followed expression becoming the declared type which is in the cast when the operation is been performed.

I/O OPERATIONS: Embedded microcontrollers must interact directly with other hardware. Therefore, any of their input and output operations are accomplished using the built-in parallel ports of the microcontroller.

DDRB = 0xff;

PORTB = z + 1;

Bitwise Operators: perform functions that will affect the operand at the bit level. These operators work on non-floating point operands: **char**, **int**, and **long**. The table below lists the bitwise operators in order of precedence.

Ones Complement	~
Left Shift	<<
Right Shift	>>
AND	&
Exclusive OR	^
OR (Inclusive OR)	

2.2.2 Control Statements

The specific control statements will be introduced in this section. It contains some common loops which are used many times in the project.

WHILE LOOP: is one of the most basic control elements, the format of this statement is as follows:

while (expression)


```
{  
    Statement1;  
    Statement2;  
    ...  
}
```

The expression is tested whether it is true or not. If the result is true, the statements in the while loop are executed. If it is false, nothing will happen.

DO/WHILE LOOP: is very similar the **while** loop, except that the tested expression is placed in the end. This means that the instructions in a **do-while** loop are executed at least one time. Similarly to *while loop*, if the expression returns true value, the loop continue working. The format of the **do/while** statement is as follows:

```
do  
{  
    Statement1;  
    Statement2;  
    ...  
} while (expression);
```

FOR LOOP: A **for** loop is used to execute a statement's block as the user knows how many times the loop should run. A variable or an expression in the for-condition braces can be initialized, test, and do an action that leads to the satisfaction of that test. The format of the *for-loop* statement is as follows:

```
for (expr1; expr2; expr3)  
{  
    statement;  
    statement1;  
    ...  
}
```

IF/ELSE statements: they are used to steer or branch the operation of the program based on the evaluation of a conditional statement.

```
If (expression)  
{  
    Statement1;  
    Statement2;  
    ...  
}
```

SWITCH/CASE statement: it is similar to *if/else statement* except that there are more values of the expression for selection. In each case, there is “*break*” command at the end to get out of the switch/case statement. The form of this statement is as follows:

```
switch (expression)
{
    case const1:    statement1;
                  statement2;
                  break;
    case const2:    Statement3;
                  ... ;
                  Statement x;
                  break;
    case const-x:   statement5;
                  statement6;
                  break;
    default:        statement7;
                  statement8;
                  break;
}
```

2.2.3 Pointer, Arrays and Structures

This section describes some kinds of compound variables, which are derived from the basic variables types. They are pointers, Arrays and Structures.

POINTERS: those variables point to the address or location of a variable, constant, function, or data object. A pointer is declared by the indirection or dereferencing operator (*):

```
char *p; // p is a pointer to a character

int *fp; // fp is a pointer to an integer
```

The pointer holds the address of another item. And in typical microcontroller the address of a memory is a 16-bit value data type, therefore the pointer will be a 16-bit value.

ARRAYS: An array is a data set of a declared type, arranged in order. An array is declared with the array name and the number of array elements x. The name of elements in array follows the syntax: name[0], name[1], ... name[x-1] .

```
int digits[10]; // this declares an array of 10 integers
```

MULTIDIMENSIONAL ARRAYS: a multidimensional array is an array of array. It can be constructed to have two, three, four, or more dimensions. The adjacent memory locations are always referenced by the right-most index of the declaration. A typical two-dimensional array of integers would be declared as

```
int two_d[5][10];
```

STRUCTURES: A structure is a compound subject created from one or more variables. The variables within a structure are called members. Members can be different types of variables.

A structure declaration has the following form:

```
struct structure_tag_name
{
    type member_1;
    type member_2;
    ... ..
    type member_x;
} structure_var_name;
```

The member can be invoked by the syntax: *structure_var_name.member_1*. The structure is a useful tool, which can manage a subject with compound features.

The specific knowledge about Embedded C which is used in the project is introduced concisely. They provide the foundation to understand the sections latter.

2.3 FreeRTOS

FreeRTOS is a real time operating system for embedded devices which is provides free by Real Time Engineers Ltd. The scheduling algorithms of FreeRTOS are designed to allow users to run multiple applications simultaneously without the microcontroller becoming unresponsive [Barry, 2009]. It also gives methods for mutexes, semaphores and software timers with expecting the embedded system would behave in a time period. If the responses is out of preferred time limit, but the microcontroller does not render useless, it is called “soft real time” behavior. If the microcontroller response are required to accomplish within a given time limit, it is so-called the system has “hard real time” behavior [Barry, 2009]. If the time limit is violated, the failure of the system would make a serious problem. Most embedded systems work in mixing of both hard real time and soft real time. The structure and working principal of FreeRTOS would be described in the following section. However, this section only gives the primitive idea of FreeRTOS.

2.3.1 Introduction

In order to manage the microcontroller cores, the FreeRTOS built Application Programmable Interface – API functions, which assigned specific duties. Those API functions are grouped to five main sets [Barry, 2009].

The first set is to manage the Task entities; the second one is to manage the Queue entities; the third one is responsible for Interrupt management, the fourth one is in charge of Resource Management and the last set is to manage the Memory. The structure of first four sets would be described in detail here.

- **Task Management**

The basic element in FreeRTOS is task, which is implemented as C function. In the task prototype there must return a void and take a void as parameter. The tasks in FreeRTOS are not allowed to return any value and must run forever. It can be more than one task in a project; therefore if the microcontroller core is “running” one task, the other tasks should be “*Not Running*”. This implies that a task can exist in one of two basic states, Running and Not Running. As a task in “*Running*” state the processor is actually implement the task’s code. When a task in the Not Running state, the task is dormant, its status has been saved for the next execution. The figure below shows present the relation between *Not Running* State and *Running* State.

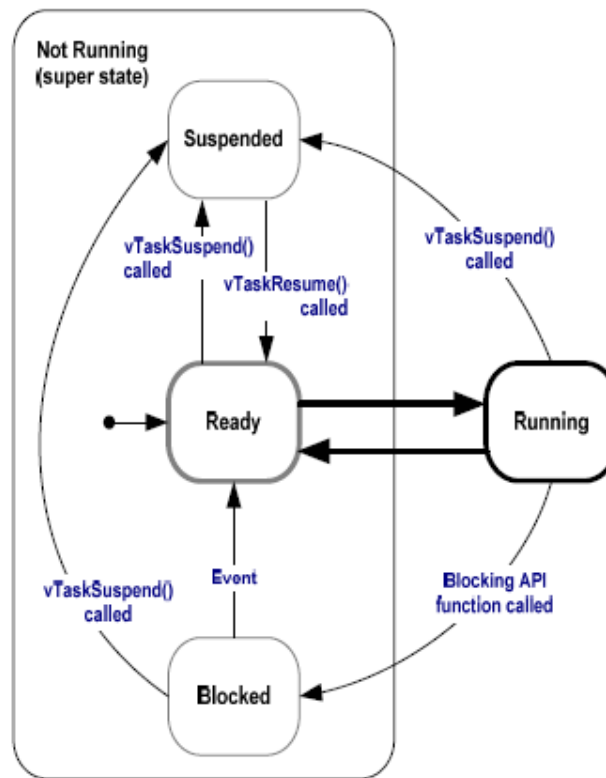


Figure 2.8 Task states in FreeRTOS

In the Not Running state, it can be separated into three sub-states: Suspended, Ready and Blocked [Barry, 2009]. A task is in the Blocked State when it is waiting for an event. The event can be temporal or synchronization one. A task is in the suspended state if it is not available to the scheduler. To get in this state, the *vTaskSuspend()* API function is called. To get out of the state, there are two way: *vTaskResume()* or

`xTaskResumeFromISR()` API functions. If a task is in the “Not Running” state but not in the Suspended or Blocked states, it would be in Ready State.

A task moves out of Not Running state to Running state is said to have been “switch in” or “swapped in”. In the opposite direction it is called “Switch out” or “Swapped out”. The FreeRTOS scheduler is the only entity that is responsible for a task switch in or out.

FreeRTOS provides some API functions to deal with the task and change the task states. In order to create or delete a task, there are two functions: `xTaskCreate()` and `vTaskDelete()`. To delay a main task in a specific time, FreeRTOS provides two functions `vTaskDelay()` and `vTaskDelayUntil()`. And to suspend or resume a task to be available for scheduler, there are two another functions `vTaskSuspend()` and `vTaskResume()`. All the API functions which are used in this master thesis project are listed on the following table:

API Function	Description
<code>xTaskCreate()</code>	Create a new task and add it to the list of tasks that are ready to run.
<code>vTaskDelete()</code>	Delete a task
<code>vTaskDelay()</code>	Delay a task for a given number of ticks; push the task to the state ‘Blocked’ state.
<code>vTaskDelayUntil()</code>	Task remains in exact specified number of ticks in ‘Blocked’ state. (Absolute time)
<code>vTaskSuspend()</code>	Suspend any task, and push the task to ‘Suspended’ state
<code>vTaskResume()</code>	Resumes a suspended task, push it to ‘Ready’ state

Those API functions above are described in primitive way. In reality, they contain more parameters which affect to the task performance.

- **Queue management**

Applications structure which applies FreeRTOS usually consists of many independent tasks. Each of them is responsible for a mini program with its own right. It can be said that the application is a collection of autonomous tasks which should communicate with each other to implement the given duties and make the entire system working properly. How the tasks can communicate and synchronize each other? Queue is a useful answer for the question.

To implement its functionality, the queue is able to store Data, is accessed by multiple tasks, block tasks on the Queue Reads or Writes. A queue can hold number of data items, the number of possible stored data is called the length of queue. The size of data depends on the kind of stored data, which can vary. The data size and queue length are set when the queue are created. A queue is able to be written and read by multiple tasks. When a task tries to read or write data on the queue, it will be blocked if the data on the queue is not available or the queue is full.

In order to create the queue, send or get the data on the queue, FreeRTOS provides a set of API functions. The table below presents a part of them, which are deployed in the project.

API Function	Description
xQueueCreate()	Create a Queue
vQueueDelete()	Delete a Queue
xQueueSend()	Post an item on a queue
xQueueSendToFront()	Post an item to the front of a queue
xQueueReceive()	Receive an item from a queue.
xQueuePeek()	Receive an item from a queue without removing the item from the queue. This macro must not be used in an interrupt service routine
xQueueSendFromISR()	Post an item into the back of a queue. It is safe to use this function from within an interrupt service routine.
xQueueReceiveFromISR()	Receive an item from a queue. It is safe to use this function from within an interrupt service routine.

Those API functions above are described in primitive way. In reality, they contain more parameters which affect to the task performance.

- **Interrupt management**

Embedded real time systems always have to response to many kinds of events. The events can be temporal/synchronization event or can originate from environment. For first kind of event, as presented earlier, the API functions or the specific task would be built to deal with them. The second kind of event is more compounds to solve. Firstly, in order to detect the event, interrupts are normally deployed [Barry, 2009]. Then whether the entire process should be in ISR or not? If the ISR is desirable as short as possible, the process should be divided. The last issue is to identify the communication between the ISR and the main code. The strategies which solve those issues would be present in following.

When an event happens, in order to synchronize a task with an interrupt, a binary semaphore can be used to unblock the task each time the interrupt occurs. That structure allows the majority code will be processed on the main task, only some critical codes lies on the ISR. Each time the event occurs, the ISR send the “give” operation on the semaphore to unblock the respective task so the required event processed is able to proceed. After using a “take” operation to run and complete its process, the task will go to Blocked state to wait for the next event.

The API functions which create a semaphore, a “give” and “take” operation are list on the table below. Semaphore can be understood just as a queue with the length is only one item. Two functions xSemaphoreTake() and xSemaphoreGive() are able to make the semaphore empty or full. The API functions which are used in this project are shown on the table below.

API Function	Description
vSemaphoreCreateBinary()	Function that creates a binary semaphore
xSemaphoreTake()	Macro to obtain a semaphore.
xSemaphoreGive()	Macro to release a semaphore.

- **Resource Management/ Kernel Control**

This part introduces how the FreeRTOS manages the resource of microcontroller. The resource management becomes critically if there is a conflict in a multitasking system [Barry, 2009]. For example, as one task accesses a resource, it is pre-empted by another task. The first task has to leave the resource in an inconsistent state then the second task access the same resource. That could make data corruption or error. In order to avoid that problem, FreeRTOS provides some API functions which disable all interrupts or scheduler or both to make a critical functions happening without intervention. Those functions are shown on the table below.

API Function	Description
vTaskStartScheduler()	Start the Scheduler
taskENTER_CRITICAL()	Macro to mark the start of a critical code region. Preemptive context switches cannot occur when in a critical region.
taskEXIT_CRITICAL()	Macro to disable all maskable interrupts.

- **Task Utilities**

This part introduces two functions which are used in the project. The two functions return the time point and the handle of running task.

API Function	Description
xTaskGetTickCount()	This function returns the count of ticks since vTaskStartScheduler was called.
xTaskGetCurrentTaskHandle()	This function returns the handle of the currently running (calling) task.

The basic concepts above show the FreeRTOS structure. It also describes some API functions which are deployed in the project. The next part would introduce some basic classes which contain the function above. Those classes are the foundation which helps FreeRTOS working properly.

2.3.2 Used FreeRTOS Library Classes

This section introduces the files which are deployed in this project. There are five code files: heap_1.c, list.c, port.c, queue.c, tasks.c and eleven header files: compiler.h, list.h, FreeRTOS.h, FreeRTOSConfig.h, io.h, portable.h, portmacro.h, projdefs.h, queue.h, task.h, StackMacros.h, semphr.h and global.h. Most of them are the FreeRTOS base which would manage the system working in the real time. Now, the function of each class would be introduced in detail following:

- *Compiler.h*

This header file defines new types of variables, macro, and constants which are deployed in the entire FreeRTOS. The new definitions help the user easily to understand as declaring new variables.

- *io.h*

This file is the interface which detects what kind of microcontroller which is connecting on the port. As being identified the microcontroller, Atmel studio would upload the code in the appropriate way.

- *heap_1.c*

This class is simply to manage microcontroller memory. The file would setup the correct byte alignment mask for the defined byte alignment and allocate the memory for the heap.

- *list.c and list.h*

Two files have responsibility to make the list of implementation used by the scheduler. While it is tailored heavily for the schedulers needs, it is also available for use by application code. Those files contain some tasks which initialize a list to store pointers or Items. The tasks is also added or removed the item to or from the list.

- *Port.c and portable.h*

Those files contain some macro and tasks which manage the switch task duties and the time tick of FreeRTOS time.

- *FreeRTOS.h*

This header file's duty is to check all the definition of required application specific macros. If there is anything missing, it would define the missed thing and make the system continuous working.

- *FreeRTOSConfig.h*

This header file configures the application specific definition which keeps the FreeRTOS working as desired. These definitions are able to adjust for particular hardware and application requirements.

- *Projdefs.h*

This class defines the prototype to which task functions must conform.

- *queue.h and queue.c*

Those classes manage the queues which are used in this project. Those are responsible to create a queue, store, send and receive data on the queue. Items are queued by copy, not reference.

- *StackMacros.h*

This header file call the stack overflow hook function if the stack of the task being swapped out is currently overflowed, or look like it might have overflowed in the past.

- *tasks.c and task.h*

Those classes declare the necessary function and feature of the tasks to manage them. They contain some tasks or macros which are able to create or delete a task, force a context switch, disable all maskable interrupts, mark the start or end of a critical code region. Preemptive context switches cannot occur when in a critical region.

- *Portmacro.h*

This header file declares Port specific definitions (new kind of variables or macro). The setting in this file configures FreeRTOS correctly for the given hardware and compiler.

- *Portable.h*

It declares the functions which are used for another class (class heap_1.c, port.c). Each function must be defined for each port. Some functions in this class are able to set up the stack of a new task or hardware/ISR so it is ready to be placed under the scheduler control, or to map to the memory management routines required for the port.

- *semphr.h*

This class declares some functions or macros which are responsible to manage the semaphore such as create/delete a semaphore, add or remove the data from semaphore. Semaphore.h is a very useful tool to manage the time trigger of scheduler.

- *global.h*

This class is different from all of the classes above because it is not from FreeRTOS library. It is created to manage the global variables, which would be used in the entire project, for example in the SEN-uC there are some variables which are used through multiple class such as Pendulum_axis, Arm_axis. They were declared in this file. Each microcontroller contains this global class but the global variables are different inside.

2.4 CAN and Time Trigger CAN

The Controller Area Network – CAN is a communication protocol, which are deployed widespread in many industrial disciplines such as automotive industry, industrial automation. The protocol has some features:

multi-master capability, broadcast communication, sophisticated error detecting mechanism and re-transmission of faulty messages [CAN in Automation, 1992]. In order to improve the bus efficiency and avoid message conflict on the bus, however, The Time Trigger CAN – TTCAN was developed. The operation on the TTCAN protocol is triggered by time scheduler, which helps transmitting messages with a lower latency jitters and guarantee a deterministic communication even at maximum bus load. The TTCAN protocol is used in this master thesis project. The detail of both protocol are introduced in the following section.

2.4.1 Controller Area Network – CAN

The Controller Area Network (CAN) protocol is an ISO standard (ISO 11898) for serial bus communication, which developed by BOSCH in 1980. It is designed for real time, broadcast protocol with a very high level of security [CAN in Automation, 1992]. The CAN standard includes the data link layer and physical layer of the ISO/OSI Reference Model. The CAN is able to transmit and receive four kinds of frame, Data Frame, Remote Frame, Error Frame and Overload Frame and achieves a bitrate of 1 Mbit/s. Those frames are identified by using a frame identifier, and the identifier has to be unique within the whole network. The identifier also defines the message priority. The lowest binary number of identifier has the highest priority. If two frames are transmitted at the same time, the conflict is resolved by bit-wise arbitration on the identifiers involved by each node observing the bus level bit for bit. It means the dominant state overwrites the recessive state, and the node which loses the arbitration becomes receivers of the message with highest priority. The lost message would be sent again when the bus is available. It can be seen that the frame is the critical mean for microcontroller is able to communicate to each other. The next section would introduce the features of CAN frame/message.

- **CAN frame**

The unit which is sent via CAN Bus is the CAN frame. The CAN protocol supports two message formats, the only essential difference is the length of the identifier. The first one is the “CAN base frame” supports identifier with eleven bits length, and second one is the “CAN extended frame” supports identifier with twenty-nine bits length [CAN in Automation, 1992].

- *CAN base frame*

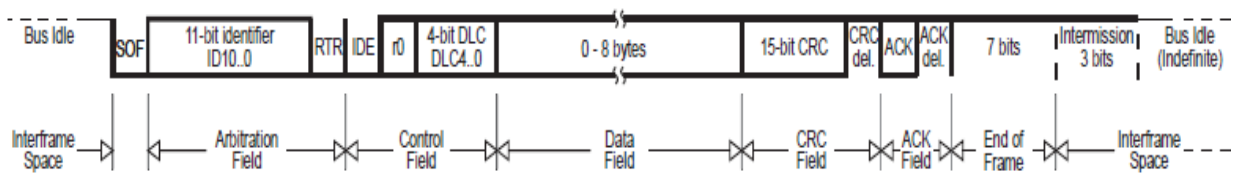


Figure 2.9 CAN message frame structure

A CAN base frame message starts with the first bit called “Start of Frame (SOF)”. The next is the “Arbitration field” which consists of the identifier as well as the “Remote Transmission Request (RTR)” bit which is used to distinguish the Data frame or the Remote frame. The third component is the “Control field” contains the “Identifier Extension (IDE)” bit to distinguish between the CAN base frame and the CAN extended frame, and the “Data Length Code (DLC)” used to the length of data byte field. If it is the Remote frame, the DLC stores the requested data byte length. The following “Data Field” contains the data up to 8 bytes data. The “Cyclic Redundant Check” that follows with 16 bits guarantees the frame integrity. The penultimate part is the “Acknowledge (ACK) field” contains the ACK slot and the ACK delimiter. The bit ACK is sent as a recessive bit and is overwritten as a dominant bit by those receivers, when the data is received correctly. The last part of the frame is the “End of Frame (EOF)”. The “Intermission frame Space” is the minimum number of bits separating consecutive messages.

- CAN extended frame

In the CAN extended frame message, there are more 18 bit extension identifier added to the 11 bit base identifier, therefore the new identifier has total 29-bit identifier used. That is the big difference between an extended frame format and a base frame format message. The IDE bit is used to distinct the CAN base frame and extended frame. If it is the 11 bit identifier frame, IDE bit will be transmitted as dominant and transmitted as recessive in case of a 29-bit frame. On the bus which operates with both kinds of frame format the 11-bit message always has priority over the 29-bit message [CAN in Automation, 1992].

The extended format however has some drawbacks: The bus latency time is longer (in minimum 20 bit-times), require more bandwidth for messages in extended format (about 20%), and the error detection performance is lower [CAN in Automation, 1992].

2.4.2 Time Trigger CAN – TTCAN

The arbitrating mechanism of CAN protocol ensures that all frames are sent according to their priority and the highest priority message will not be disturbed. It means that the lower the priority of a message is, the higher the latency jitter for media access. In order to reduce the latency jitters and guarantee a deterministic communication pattern even at maximum bus load, the Time triggered CAN – TTCAN protocol was

developed to meet the requirements. Time trigger communication means all events, transmitting or receiving in the network, is activated by the time segment elapsing. The TTCAN protocol is extended for time triggered execution of CAN within ISO 11898-4 in two levels [Thomas Führer]. In the level 1, the time triggered operation of CAN is guaranteed by the reference message of a time master. Meanwhile, the CAN Operation of the second extension level is triggered by a global synchronized time and the time drift correction is realized. The TTCAN communication is deployed in this project based on AT90CAN128 controller, which support the extension level 1. Therefore, the later sections would focus on presenting the specification of the TTCAN level 1 protocol.

- **Reference message**

TTCAN operation is based on the time trigger and periodic communication which is originated by the reference message from the time master. The reference message is distinguished to another message by its identifier. This message is sent on time slot “0” on the basic cycle (which is introduced in latter section) and contains the information of the transmitted time of the reference message itself. As the reference message arrive other node, it would be used to correct the time drift between time master and others.

- **The basic cycle and its time windows**

The epoch between two consecutive reference messages is called basic cycle. The basic cycle contains several time slots which are also called time windows. The time window has different size and enough space for the message transmission [Thomas Führer].

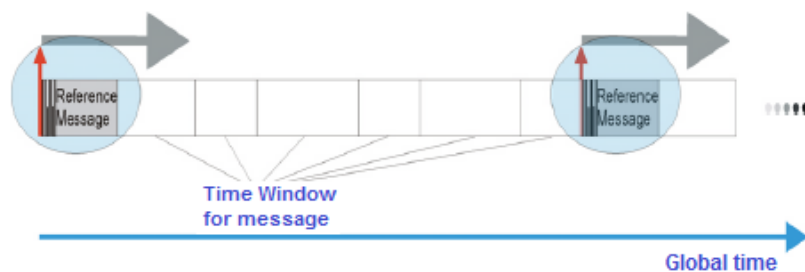


Figure 2.10 Basic cycle and Time Window

The message which is hold in the time window can be period or spontaneous state, event or data. Any message has the standard of CAN message, which is presented in the previous section. There are three kinds of time window on the basic cycle. The first one is called exclusive time window, stores a periodic message. The transmitting time point of the message on the exclusive time window is determined. The node which sends the message is also predefined. The kind of time window is designed off-line and can be more than one on a basic cycle. However, this time window does not allow to implement an automatic re-transmission of CAN message.

In order to make the design's work more flexible, the second kind of time window was developed. It is called arbitrating time window and uses for spontaneous messages. Within this time window, the CAN message is designed on-line. It means in design time, there is more than one message for this time window. The specific message which is chosen to send on the bus is arbitrated by bitwise arbitration. This time window also prohibits automatic retransmission of CAN message.

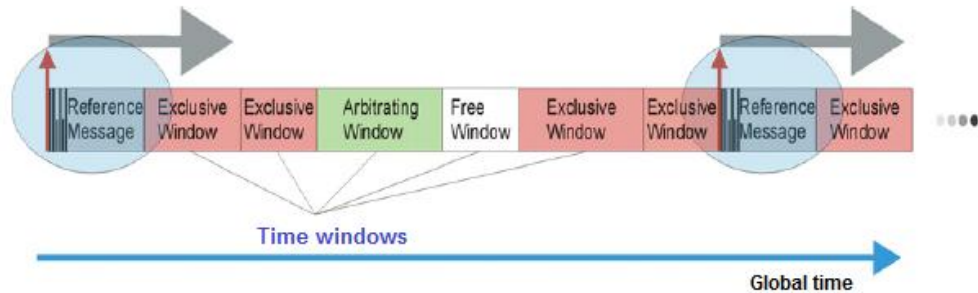


Figure 2.11 Basic cycle with three kind of time window

The last kind of time window is called free time window. It is designed to reserve for further extensions of the network. It can be changed to exclusive or arbitrating time window as the new nodes are added to the CAN system. Three kinds of time windows are shown the above figure with the reference message.

- **The nodes in TTCAN network**

In TTCAN network, a node does not have to know all the messages. It only receives the necessary information for time triggered operation such as sending or receiving messages. This feature allows optimized memory utilization in a hardware realization but offers still enough information for network management [Thomas Führer].

- **The System Matrix**

For a sophisticated system, one basic cycle could be not enough space for the communication among the nodes. In order to deal with that issue, the TTCAN specification allows using more than one basic cycle to build the communication matrix or system matrix. The figure below present a system matrix contain four basic cycles.

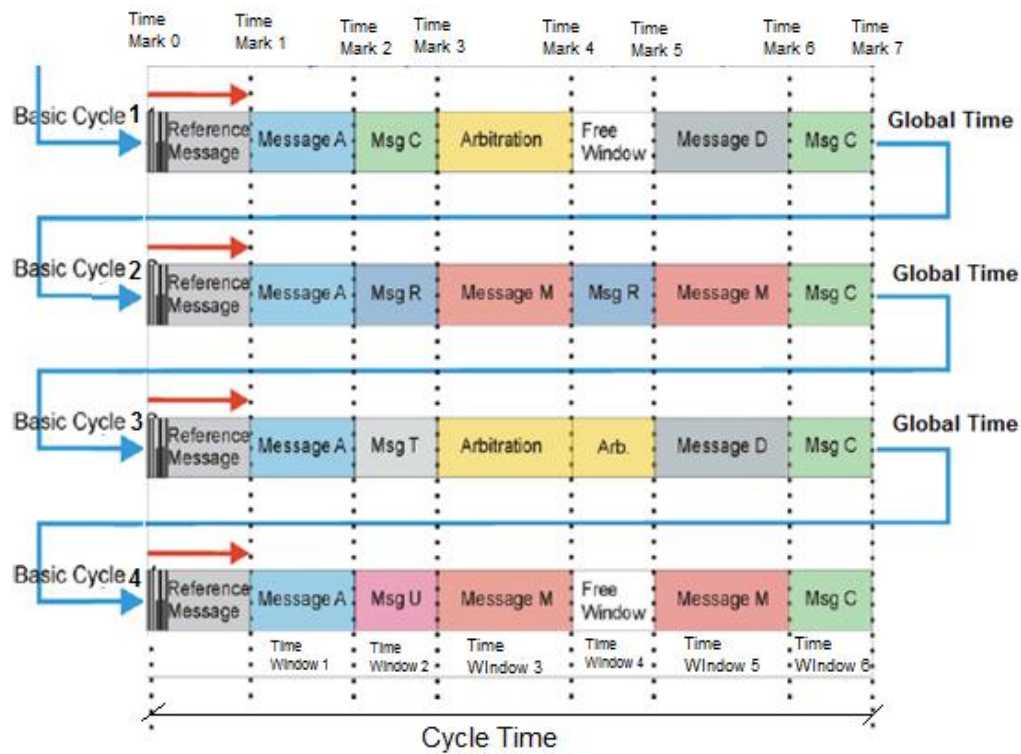


Figure 2.12: System Matrix

Four basic cycles are connected together to build system matrix. In this system matrix, a CAN message can be sent in every basic cycle, every third basic cycle or only one time in the whole system matrix.

- **Cycle Time and Time marks**

Within a CAN basic cycle, the communication is executed by the elapsing of time. The kind of time is called the Cycle Time of TTCAN and be reset after the reception of every reference messages. The system matrix is linked with the cycle time by Time Marks. They specify the starting point of each time window. Time Marks for transmitting messages are called transmitted Trigger – Tx Trigger. Similarly, Receiving Trigger – Rx Trigger is used for the message reception.

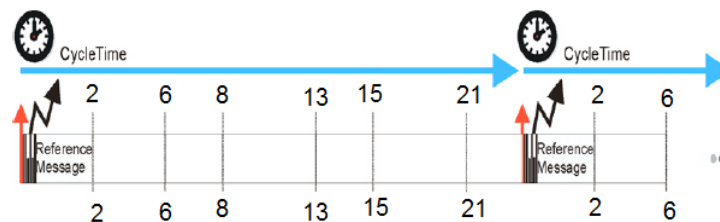


Figure 2.13 Cycle Time

If the system matrix has more than one basic cycle, the time mark would furthermore contains the base mark and the repeat count. The base mark's duty is to determine the number of the first basic cycle after beginning

of the matrix cycle in which the message must be sent or received. The repeat count determines the number of basic cycles between two successive transmissions/receptions of the message in one system matrix.

2.5 AT90CAN128 Microcontroller

In this section, the AT90CAN128 microcontroller and its CAN controller will be introduced. The microcontroller has a critical role to control the IRP working. It receives data from two encoders, calculates, processes and converts the signal to PWM to manipulate the DC-motor. Because of high requirements in IRP control, four AT90CAN128s are deployed. Each microcontroller is assigned a specific duty then the result from the microcontroller's processes would be exchanged via CAN controller. Because of the complicated structure of the AT90CAN128, only those parts which are used to in this master thesis project are introduced here. This section is divided into two parts. The first one introduce about the specific knowledge about AT90CAN128. And the second one presents the CAN communication which is attached on it.

2.5.1 The specific knowledge about AT90CAN128

AT90CAN128 is a powerful microcontroller of Atmel Corporation, which is based on 8-bit enhanced RISC architecture. It can achieve the calculation speed approaching 1 MIPS per MHz, CPU clock frequency is 16MHz and support strongly for multiple applications by large memory, 32 general purpose working registers, ISP and USART communication [Atmel Corporation, 2008]. In this project there is a part of them applied in this project. The overview of this microcontroller is shown in the figure below.

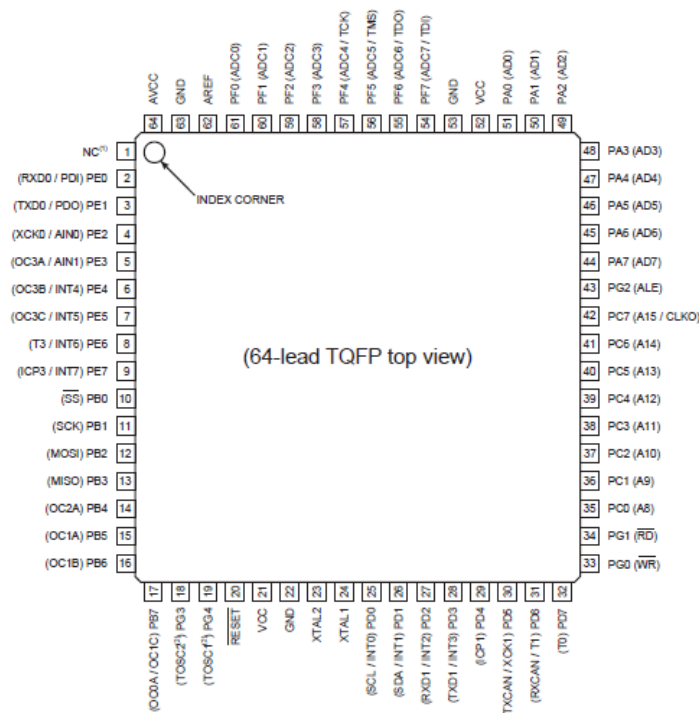


Figure 2.14 AT90CAN128 Top view

- **I/O ports**

The parallel I/O ports are the most general-purpose I/O devices. There are three kinds of register involving with the parallel I/O ports. The Data direction register, DDRx is to control the Pin of I/O ports are input or output (where x is the name of a specific port). The second one, the *port driver register* – PORTx, is used to control the voltage level of the Pins. The last register is PINx. It saves the states of each Pin of the Ports.

There are 7 I/O ports with 53 general purpose I/O lines on AT90CAN128. All the ports have true Read-Modify-Write functionality when used as general digital I/O ports. This means the direction of one port pin can be changed without unintentionally changing the direction of any other pin. The same rule applies when changing drive value or enabling/disabling of pull-up resistors. Besides that, those I/O ports can work as communication ports (SPI, USART, CAN), ADC port or receiving external clock.

- **Interrupt**

An interrupt is a signal which causes the CPU to stop the current operation then jumps to a specific code which is designed to response the signal. The signal is generated by an external events or internal hardware source. The specific invoked code is called interrupt service routine (ISR). Upon the completion of the ISR, the program flow will resume and continue from where it was interrupted. Interrupt is a powerful tool which can make the embedded system more efficient and responsive the critical events.

When an Interrupt happened, first the program counter and some possible status information is saved by the CPU, the ISR is executed. When the ISR complete, there will be a return-from-interrupt instruction which will restore any automatically-save status information followed by the saved program counter. Interrupts in each microcontroller are grouped to a list of fixed address called vectors. If an interrupt D is active on a microcontroller, the fixed vector address of that interrupt, call it address AD is saved in the microcontroller. To response to the interrupt, an ISR is written and located in memory starting at address AISR. Then the vector for interrupt D (the memory starting at address AD) will either contain an instruction jump-to-AISR or just consist of the address AISR.

Normally, an ISR has the higher priority than ordinary task. The ISR can finished when it completes it command and just returning a return-from-interrupt instruction. Meanwhile, the ordinary task is designed to run forever. If there is no data to work, the task will go to block state.

AT90CAN128 support a various kind of interrupts which can work in many practical conditions. Those interrupts can be applied to detect the signal change on the Pins from environment (the external interrupt), to manipulate PWM or control the Peripheral devices (the Timer/Counter Compare Match Interrupt), to check data are sent or received completely (The Tx Complete or Rx Complete Interrupt). In this IRP project, the

external ISR, Timer/Counter ISR and CAN Transfer Complete ISR are critical components to achieve the project goals.

- **Timer/Counter**

As the name is called, timer/counter counts the elapsing time with the scale of CPU timer or external timer. It is a high versatile tool, is able to measure time periods, determine pulse width, measure speed and frequency, or to provide output signals. There are four timers/counters (timer0, timer1, timer2, timer3) on this AT90CAN128. Timer/Counter0 and Timer/Counter2 are the 8 bits counter with single Channel Counter and which can used to control PWM. Those Timer/Counter are not applied in this project.

The others are 16 bit Timer/Counter which allow accurate program execution timing, wave generation and signal timing measurement. Both timers are controlled by the 8-bit Timer/Counter Control Registers TCCR1/3, which set up the Prescaler and the Waveform Generation Mode. The 16 bit Timer/Counter Register TCNT1/3 can be clocked internally via the prescaler. The Timer/Counter incorporates an Input Capture Unit (ICR1/3) or Output Compare unit (OCR1/3) those can capture events or compared match and give them a time-stamp indicating time of occurrence. All interrupts are individually masked with the Timer Interrupt Mask Register (TIMSK1/3), and visible in the Timer Interrupt Flag Register (TIFR1/3). The combination of the Waveform Generation Mode and the Compare Output mode bits define some behaviors of the Timer/Counter and the Output Compare pins, which are called the mode of operation. The operation mode is applied in Pulse Width Modulation.

- **Pulse Width Modulation (PWM)**

PWM is a method to convert the digital signal to analog one. In this method, the duty cycle of the square wave output from microcontroller pins are varied to supply the varying DC voltage output. The effective output is the average of DC voltage. The variation of the output voltage is created by the combination of Waveform Generation Mode and the Compare Output mode bits. The combination creates some kind operation mode which can vary the output energy and frequency. Based on the application requirements, user can choose one of the operation mode such as Normal Mode, Clear Timer on Compare Match Mode, Fast PWM Mode, Phase Correct PWM Mode or Phase and Frequency Correct PWM Mode.

2.5.2 CAN Controller in AT90CAN128

The CAN controller which is implemented in AT90CAN128 offers V2.0B Active. There is a component called Message Object (MOB) attached to the CAN controller, which store all information regarding the message. That component supports the full-CAN controller to be convenient for acceptance filtering and

message management. In the peripheral initialization phase, the application defines which messages are to be sent and which are to be received. The CAN controller only receives the message whose identifier matches with one of the identifiers of the programmed (receive-) message objects. Then the accepted message is stored and the application program is informed by interrupt. Another advantage is that incoming remote frames can be answered automatically by the CAN Controller with the corresponding data frame [Atmel Corporation, 2008].

In order to communicate the physical bus, the CAN controller needs a CAN controller interface. In this project, the interface is performed by a device named PCA82C250. The interface device converts the RxCAN or TxCAN signal from CAN Controller to CANH or CANL signal on the physical bus and vice versa.

• CAN Controller Structures

The CAN Controller Structure contains two main components: The CAN channel and the Message Object (MOB). The CAN channel controls the communication parameters of CAN Controller with physical bus via the PCA82C250. The MOBs are responsible to handle all CAN messages. Depending on the working requirements, the Mobs are set for appropriate operation mode.

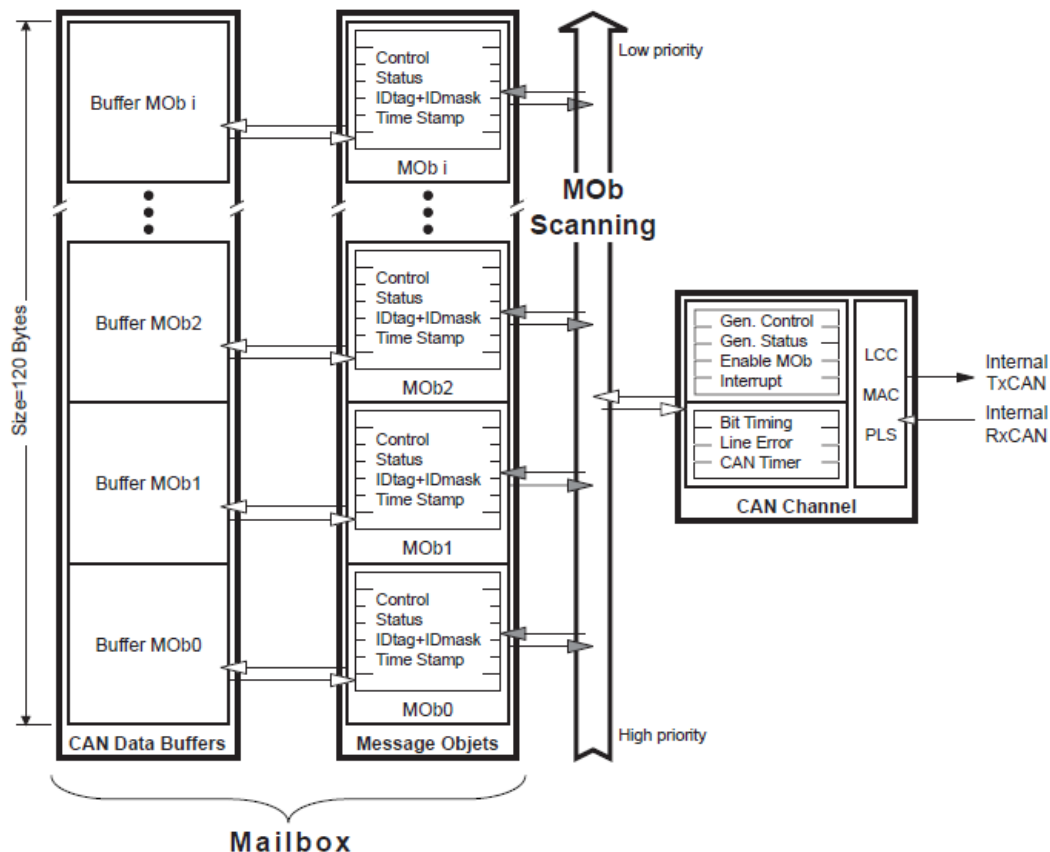


Figure 2.15 CAN controller

- CAN Channel

The CAN Channel is the bridge between the MObs and the physical bus. It can operate in some kinds of modes. The first one called the Enable Mode the CAN Channel (internal TxCAN and RxCAN) and the input clock are enabled. The second mode is Standby mode. In this mode, the transmitter constantly provides a recessive level and the receiver are disabled, but the input clock is enabled and the register and pages remain accessible. The last mode is Listening mode, which is transparent for the CAN Channel.

In order to implement communication, The Finite State machine of the CAN Channel is needed to synchronize with the time quantum. Then the baud Rate is set to choose the appropriate sampling point.

- Message Objects

The MOB is a CAN frame descriptor which contains all information to handle a CAN frame. There are 15 independent MOBs, however priority is given to the lower one in case of multi matching. They have some operation mode: disabled mode, transmit mode, receive mode, automatic reply and frame buffer receive mode. In order to receive only the appropriate message, Acceptance Filter is used to choose the fit one. The Filter is performed by configuring the registers CANID1/2/3/4 and CANIDM1/2/3/4. In order to move to the desired MOB among 15 MOBs, AT90CAN128 provides a CANPAGE register. Based on the value of the CANPAGE first four bits, the CAN controller can know which MOB it is working with. When the data is ready, and the desired MOB is chosen, those data saved on CAN Data Message register CANMSG.

• CAN Timer

CAN Timer is a programmable 16-bit timer is used for message stamping and time trigger communication.

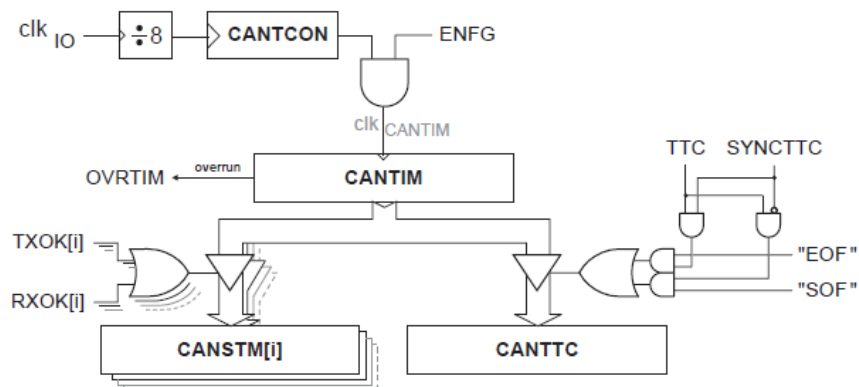


Figure 2.16: The CAN timer

As shown on the figure above, CANTCON is responsible to configure the CAN Timer prescaler. The clk_{CANTIM} is calculated by the following formula:

$$Tclk_{CANTIM} = Tclk_{Io} \times 8 \times (CANTCOM[7:0] + 1)$$

The CANTIM is a 16 bit register, counts from 0x0000 to 0xFFFF. It begins when the CAN controller is enabled and reset “0” when its value is over 0xFFFF. In the Time Trigger Communication mode, the CANTTC register save the value of CANTIM when a message arrive or is sent. The capture time value is done in the MOB which receives or sends the frame.

• CAN Interrupt

In order to check a message is sent or received completely, interrupts are applied to inform to CAN controller. Besides that, interrupts is also used to detect the transmission fault, check the MOB available and the bus state. The CAN interrupts structure is shown in the following figure:

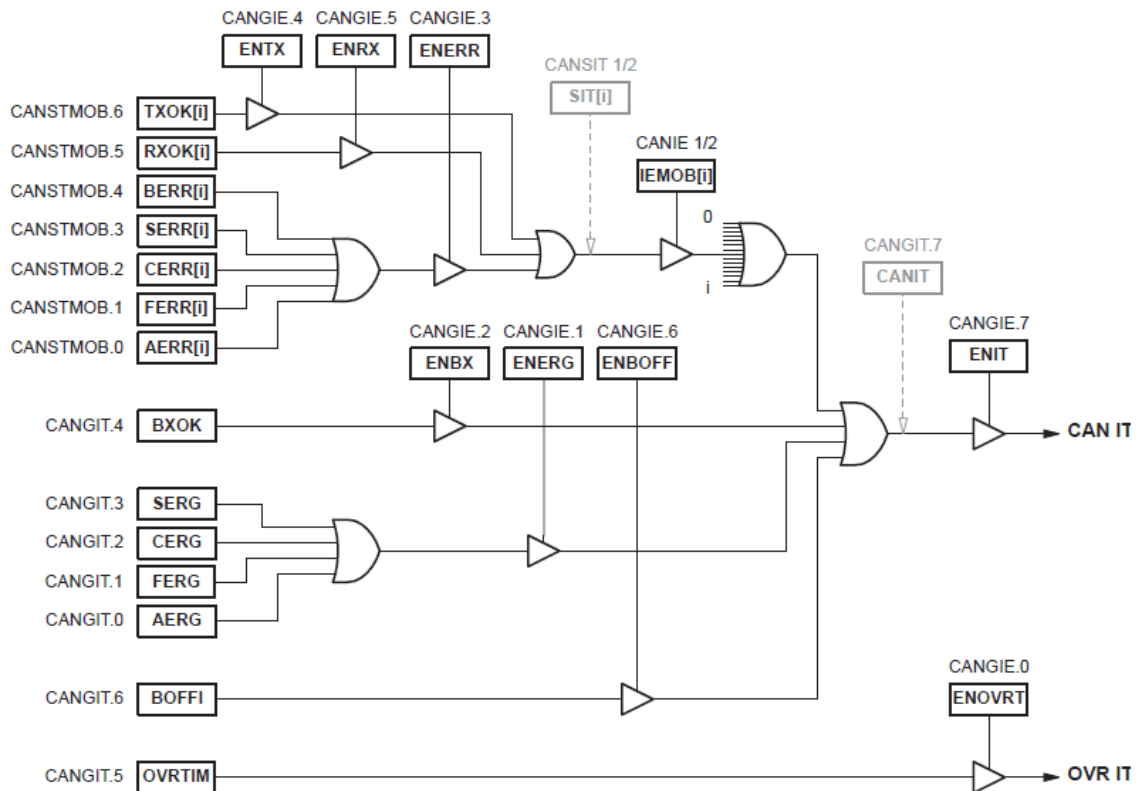


Figure 2.17 CAN interrupt structure

On the structure, there are some kinds of interrupts:

- Interrupt on receive completed OK,
- Interrupt on transmit completed OK,
- Interrupt on error (bit error, stuff error, CRC error, form error, acknowledge error),
- Interrupt on the frame buffer full,

- Interrupt on “Bus Off” setting,
- Interrupt on overrun of CAN timer.

The general interrupt enable is provided by ENIT bit and the specific interrupt enable for CAN timer overrun is provided by ENORVT bit.

3 Modeling and Microcontroller's Structures

This chapter presents the Nonlinear Model of the IRP and the structure design of the four microcontrollers which are responsible for the IRP control. The first part 3.1 fulfills the real model of the IRP from my student work. The latter section would build the microcontrollers structures. It starts with assigning the duties for each microcontroller and set the communication channel – TTCAN. Then the TTCAN scheduler is set up the basic cycle and time window. The designed work continues going to detail such as when a specific action happens by what microcontroller. And which task will be active when a message comes and what should the task do.

Let's start the interesting works by the nonlinear IRP model.

3.1 The nonlinear IRP Model

In the student work [Bui, 2013], the IRP model is a linear model. However, the IRP is nonlinear machine that means the linear model is not able to represent all features of the IRP. In order to have a better view about the IRP, constructing a nonlinear IRP model is a necessary work. In the nonlinear model, only the PLANT will change, other components such as the Feedback, Integral components still keep their structure. The nonlinear IRP model is built basing on the original equation system [Bui, 2013]:

$$\rightarrow \begin{cases} (J + m\rho^2 + ml^2 \sin^2 \phi) \ddot{\alpha} - (m\rho l \cos \phi) \ddot{\phi} + (2ml^2 \sin \phi \cos \phi) \dot{\alpha} \dot{\phi} + (m\rho l \sin \phi) \dot{\phi}^2 = M_{input} \\ (-m\rho l \cos \phi) \ddot{\alpha} + ml^2 \ddot{\phi} - (ml^2 \sin \phi \cos \phi) \dot{\alpha}^2 - mgl \sin \phi = 0 \end{cases} \quad (3.1)$$

In order to ease the modelling job the equation system (3.2) is rewritten:

$$\begin{aligned} \ddot{\alpha} &= \frac{M_{input} + \frac{m\rho \sin 2\phi}{2} (l\dot{\alpha}^2 \cos \phi + g) - ml(l\dot{\alpha} \dot{\phi} \sin(2\phi) + \rho \dot{\phi}^2 \sin \phi)}{J_{tot} + ml^2 \sin^2 \phi - m\rho^2 \cos^2 \phi} \\ \ddot{\phi} &= \frac{(J_{tot} + ml^2 \sin^2 \phi)(l\dot{\alpha}^2 \cos \phi + g) \sin \phi + \rho M_{input} \cos \phi - m\rho l \left(l\dot{\alpha} \dot{\phi} \cos \phi + \frac{\rho \dot{\phi}^2}{2} \right) \sin(2\phi)}{(J_{tot} + ml^2 \sin^2 \phi - m\rho^2 \cos^2 \phi)l} \end{aligned} \quad (3.2)$$

The nonlinear Simulink model of the IRP is shown on the figure below.

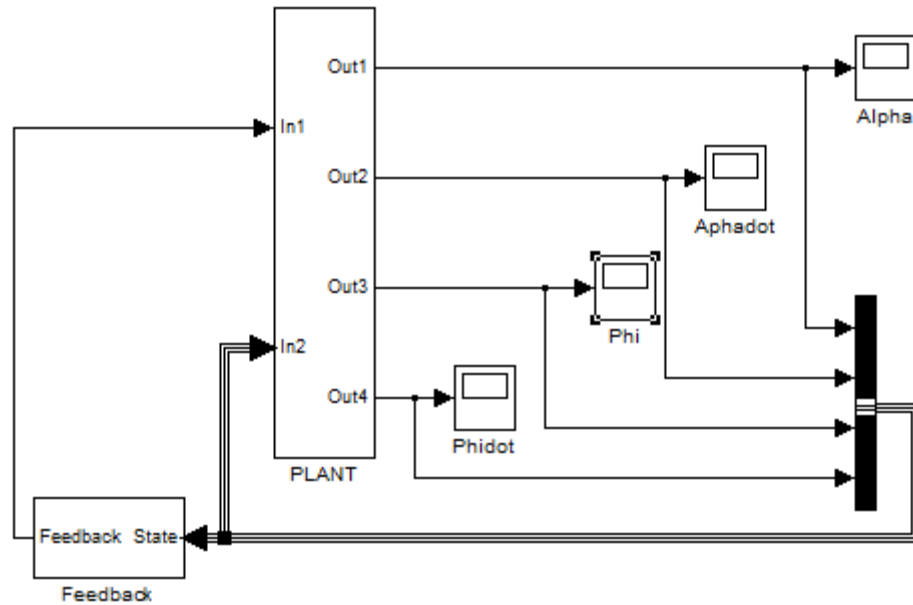


Figure 3.1: The Nonlinear Simulink IRP model with no reference input

In the PLANT component, the structure is change dramatically. Because of the complexity of the IRP machine, the PLANT component is separated to 3 subsystems, which is called the Nominator1, Nominator2 and Denominator.

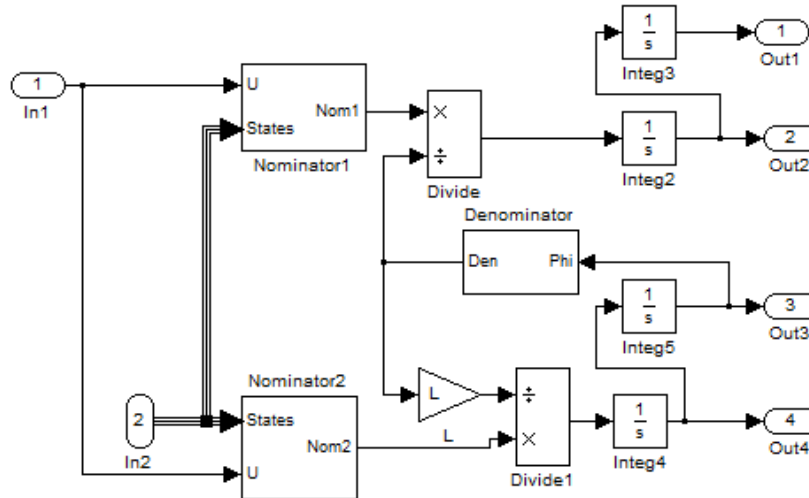


Figure 3.2: The nonlinear PLANT

The nominator1, Nominator2 and Denominator are respectively with the two nominators and denominator on the equation system (3.2). The structure details of three subsystems are then shown on the three figures below:

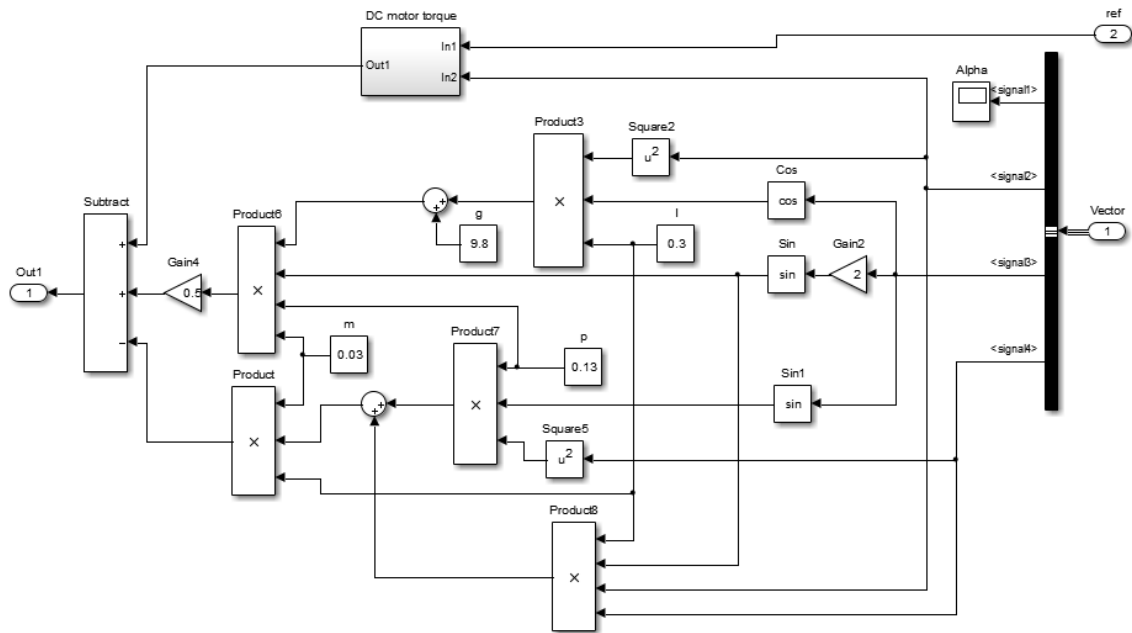


Figure 3.3: The Nominator 1 subsystem

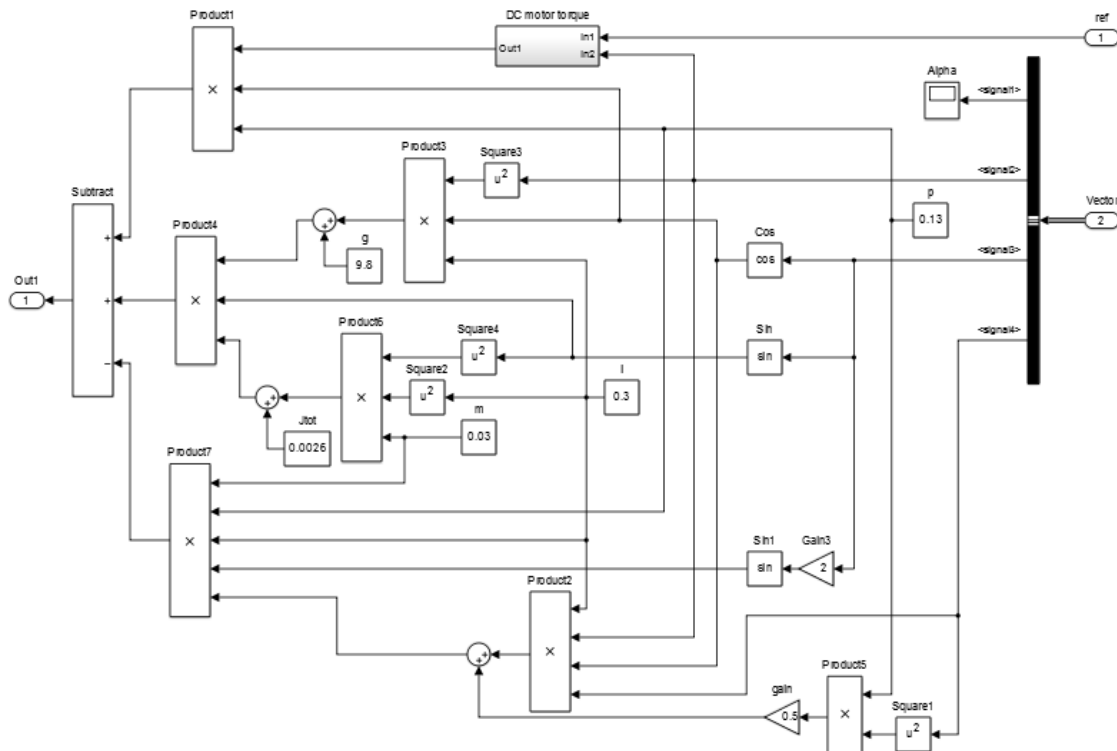


Figure3.4: The Nominator 2 subsystem

In order to reduce the complexity of the Simulink model, some IRP's parameter such as inertia momentum J_{tot} , gravity constants g , the mass m , the rod length L , the arm length r are used more than one time in the subsystems as inputted parameters.

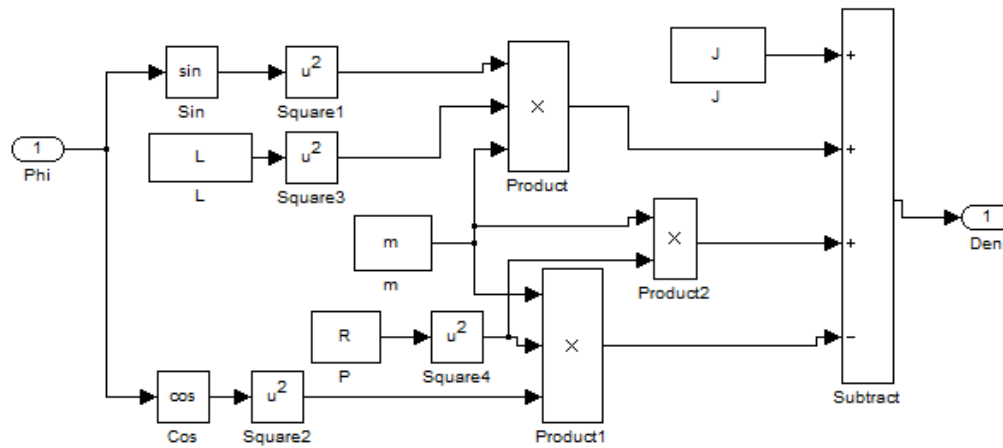


Figure 3.5: The denominator subsystem

In both nominator subsystems, there is one part which describes the DC-motor equation. Because of its compound structure, it is grouped to be another subsystem, which is called DC-motor Torque. In this Simulink model, K_t and K_m is the motor parameters, η_g , η_b , η_m is the efficiency of the gearbox, the belt transmission and the DC-motor. K_g and K_b is the gear and belt ratio.

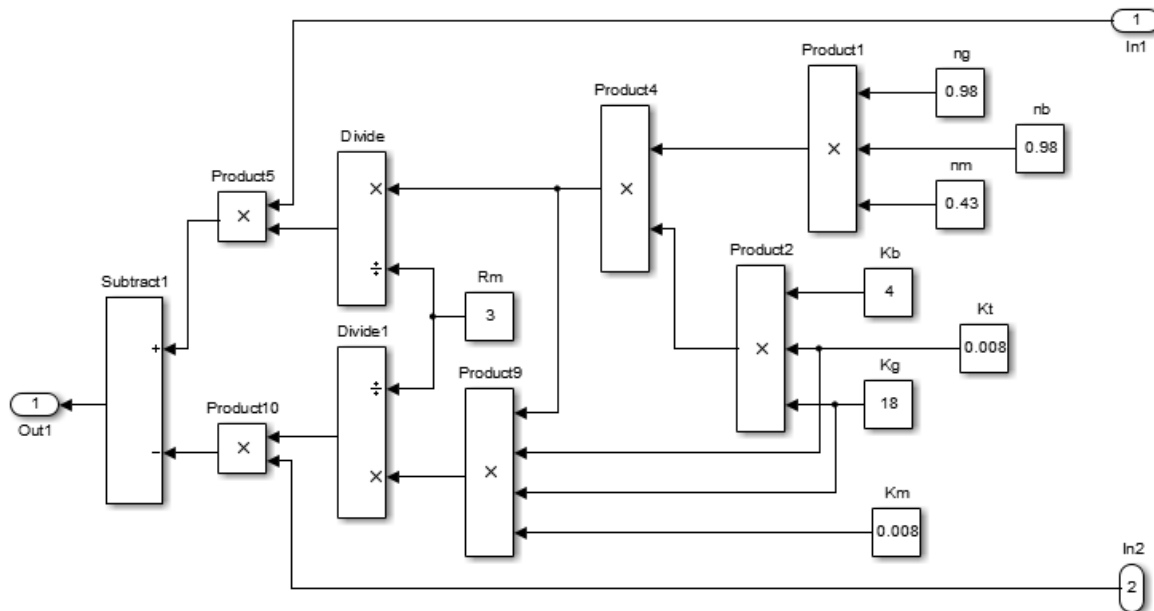


Figure 3.6 DC-motor torque subsystem

Because there is no change on the Feedback component, their structures are the same from the section 2.1.2

To improve the flexibility of the IRP, the integral part is added to the system. With the added part, the IRP is able to work at an arbitrary the arm angle. The state vector of the IRP increases one more degree becoming the vector with five states.

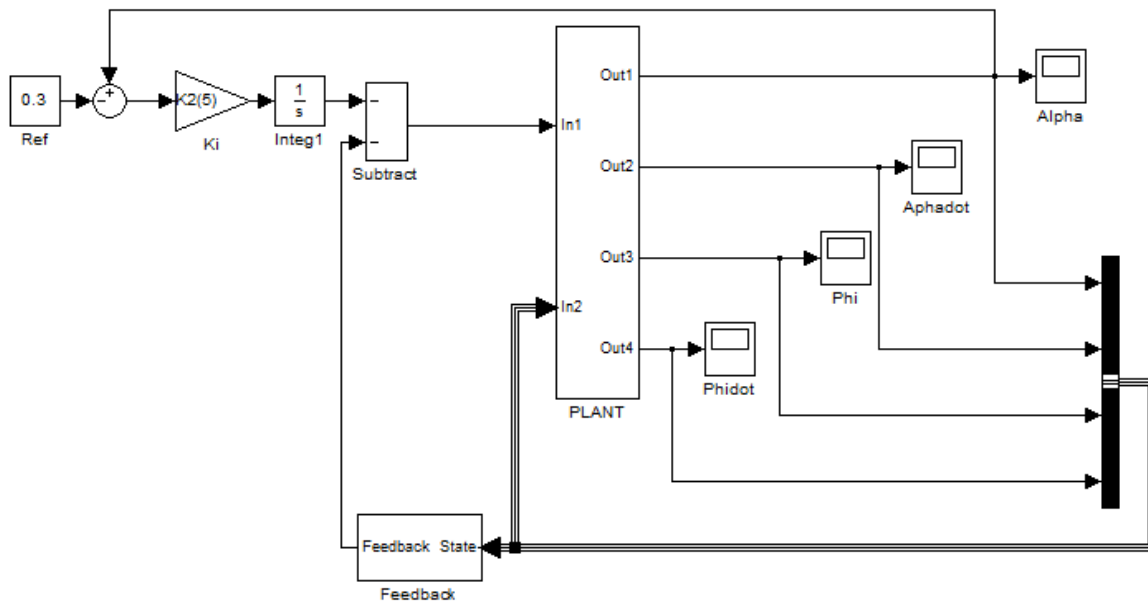


Figure 3.7: The Nonlinear Simulink IRP model with no reference input

3.2 Microcontroller Structures

There is combination of four microcontrollers to control Inverted Rotary Pendulum (IRP). Four microcontrollers are called sensor microcontroller (Sen-uC), LCD microcontroller (LCD-uC), Push Button microcontroller (Push-uC) and Motor microcontroller (Moto-uC). They are assigned separate works, as following description:

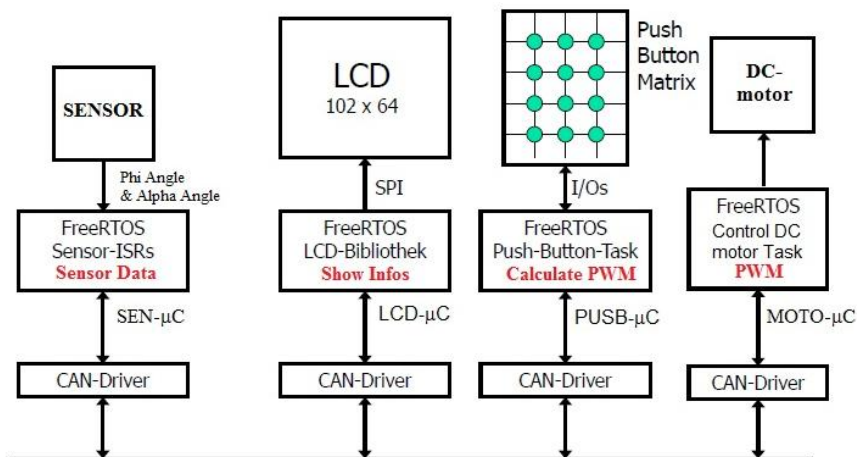


Figure 3.8 The IRP block diagram

The Sen-uC: the main duty of this microcontroller is to get the sensor signal in every three milliseconds, then transmit to all three microcontrollers. The Sensor data are got from two global variables, Pen_Ang and Arm_Ang, which are changed by External Interrupt Service Routine.

The LCD-uC: the main duty of this microcontroller is to show the system information on LCD screen such as: system state, two angles, PWM, Reference Input, Start/Stop buttons, Change the Reference Input value. It receives two CAN messages which are Sensor/Angle message and Pusb_uC message. Base on the data on two message frames, LCD-uC shows the respective information. This microcontroller is the time master of the whole system.

The Pusb-uC: the main duties are to calculate the PWM to control the DC-motor and detect whether any button is pushed or not. It receives one message frame from Sen-uC, Sensor message and that is the input of PWM calculation task. Besides that, this microcontroller senses any pushed button and transmits the result to PWM calculation task and other microcontroller by using Pusb_uC message frame.

The Moto_uC: the main work is to control DC-motor. It receives two messages which are Sensor and Pusb_uC message. With data on Angle message, Moto_uC chooses which task should be deployed. Other one, Pusb_uC will be used to control the direction and duty cycle of DC-motor.

In order to understand the detail structure of each microcontroller, the next part introduces some general information which involve to all microcontroller.

3.2.1 TTCAN Scheduler

As presented in the previous section, there are four microcontrollers involving to control the IRP. In order to make the IRP working properly, four microcontrollers must communicate together and transmit the data to each other. As presented in the section 2.4, TTCAN is deployed in this project. The first thing which is considered in TTCAN is TTCAN Scheduler. In the scheduler it presents the basic cycle duration, how many time windows and message frames, when they are sent, where do they come from, and where do they arrive. All the information is shown on the figure below.

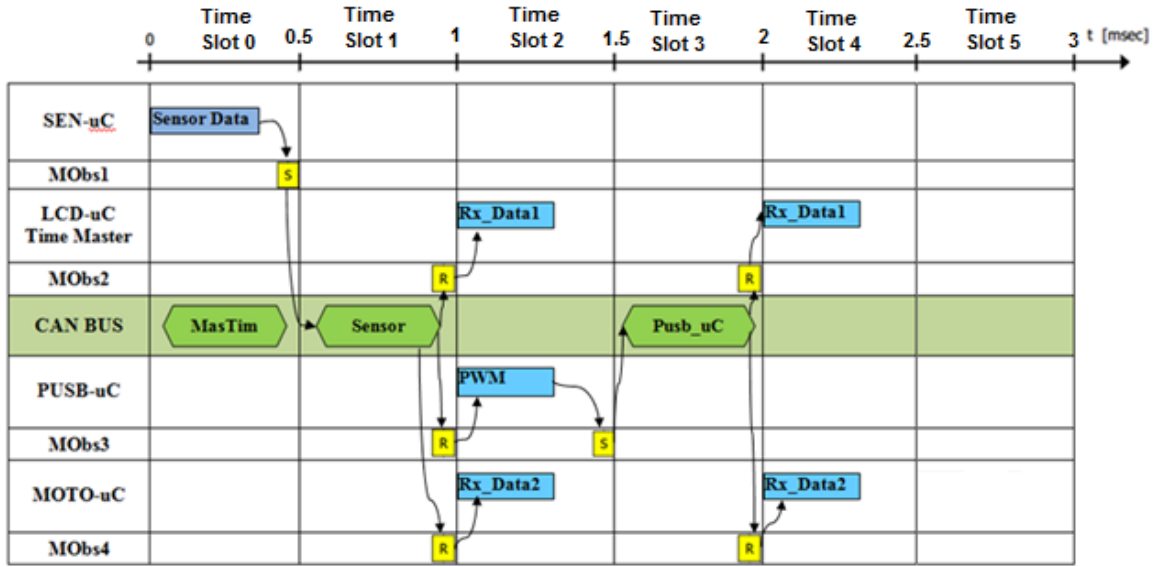


Figure 3.9 TTCAN Scheduler

To manage the IRP system, the requirements on the time response is critical. The system must detect any changing of the pendulum angle then have an appropriate response. The faster the system is, the more stable the pendulum is. In this TTCAN scheduler, there is one basic cycle and it lasts in 3 milliseconds. In that period, there are six time windows with equal period 0.5 millisecond. Those microcontrollers communicate together by three message frames. They are Reference message (MasTim) with id 0x0000, Angle/Sensor Message (Sensor) with id 0x0001 and the last message is Push button message (Pushb_uC) with id 0x0002. The first message, reference message – MasTim, is sent by LCD-uC in Time Window “0”. Its content is the transmitted time point of the last reference message. This transmitted time point is used to correct the time drifting between the Time Master and other microcontrollers. The second message – Sensor Message (Sensor) – contains the information of two angles of IRP, is transmitted by SEN-uC on Time Slot 1. The last message Pushb_uC contains PWM value and controlled buttons, which is processed by PUSHB-uC and sent on time slot 3. The contents of the messages are clear, now a structure object is needed to create to store the content. The structures must be compatible with the CAN message which is presented in section 2.4.1. Based on that, the structure should consist of some members as following:

```
struct CAN_message
{
    unsigned int id;           // MSG ID 11 bit
    unsigned char length;     // DLC Length
    unsigned char data[8];    // Data field 8 Byte
} Angle, Pushb_uC, MasTim;
```

id is the member variables to store the identifier of message, *length* variable is responsible for the data length of message and the last – an array *data[8]* store the data of the message.

- **Data length of three CAN message**

Reference message – MasTim: it contains the transmitted time of the last reference message, which is a 16-bit value. Thereby there are two bytes in data array – *data[0]* and *data[1]* which store the transmitted time.

Sensor/Angle message – Sensor: Both encoders have resolution 2000 values/1 revolution, it is necessary to have integer variables to manage the angle of the Arm and the Pendulum. To control the pendulum from the down position to the upright position, it is needed two kinds of pendulum angle. One for swing up phase is called Pendulum angle with the zero value at the downright position. The second one is to use to control the pendulum at upright position which is called Pendulum up with the coordinate at exact upright position. Therefore, there are three integer variables, one for arm angle and the others for pendulum angles. With six bytes for the angle variables, there are needed six data bytes on the message *Sensor*, they are *Sensor.data[0]*, *Sensor.data[1]*, *Sensor.data[2]*, *Sensor.data[3]*, *Sensor.data[4]*, *Sensor.data[5]*. The data length on this message is 6 bytes.

The Push Button microcontroller message – Push_uC: The PWM value is an integer value; therefore it is needed two data bytes to store it. Moreover, this message contains the controlled signal from the pushed button, so the arrangement of the data bytes is organized as following:

- The first byte *Push_uC.data[0]* stores the started button signal (Button 1 on the button matrix).
- the second byte *Push_uC.data[1]* stores the stopped button signal (button 2),
- the third byte *Push_uC.data[2]* stores the changed reference input button signal (button 3),
- the fourth byte *Push_uC.data[3]* stores the value of reference input data (button 6),
- the fifth byte *Push_uC.data[4]* and the sixth bytes *Push_uC.data[5]* store the PWM data.

The three bytes *Push_uC.data[0]*, *Push_uC.data[1]*, *Push_uC.data[2]* only store two values “0” or “1”. The value is “1” if the respective button is pushed, and is “0” if it is vice versa.

3.2.2 Time Trigger Matrices

Based on the TTCAN Scheduler, the Time Trigger Matrices are constructed to control the microcontrollers. Those Matrices control each microcontroller when and what to do a specific action. Before going to build the matrices, the overview of each microcontroller jobs will be introduced. The SEN-uC sense the changing of both encoders by ISR, save the new angles in every 3 milliseconds at time slot “0”, then transmit the data in the following time slot 1. The LCD-uC receives both messages *Sensor* and *Push_uC* on time slot 2 and time

slot 4 respectively then shows the data on the LCD. The MOTO-uC does analogous to the LCD-uC except that the data is used to control DC-motor. The last microcontroller PUSB-uC receives *Sensor* message on time slot 2, processes the sensor data and Button signals then transmits the *Pusb_uC* message at time slot 3.

In order to synchronize the four time trigger matrices, they are designed with the same structure and define values. The define value meanings are explained as following:

```
#define NONE      0
#define RX_ANGLE  1
#define RX_PUSB   2
#define PR_DATA   3
```

NONE: means there is nothing happened at the respective time slot, and its value is zero “0”.

RX_ANGLE: the microcontroller receives the Sensor/Angle message in the current time slot. Its value is “1”.

RX_PUSB: the microcontroller receives the *Pusb_uC* message in current time slot. Its value is “2”.

PR_DATA: the microcontroller prepares sensor data to send on the next time slot. Its value is “3”.

The Time Trigger Matrices are two dimensional arrays with 7 columns and 6 rows. The six rows are corresponding with six time slots (from time slot 0 to time slot 5). Meanwhile the columns are more complicated. The first, second and third column indicate respectively the *Time-mark*, *Base-mark* and *Repeat index* of the Time Trigger Matrix (Section 2.4.2). *Time-mark* is variable which manage the time windows of the schedule running from “0” to “5”. Those time trigger matrices have only one basic cycle so the *base-mark* and *Repeat* values are “0” and “1” respectively. The fourth column called *Type* indicates the actions which the TTCAN Scheduler should do. If the *type* has value such as ‘S’ or ‘s’, a CAN message will be transmitted. If the *type* is ‘A’ or ‘a’, that time slot is for the arbitrary slot, which will be used in future. If the type is ‘T’ or ‘t’, a specific task will be woke up to prepare the data. The actions in each time slot are difference depending on microcontroller’s duties. If the microcontroller sends or receives a message, the fifth column shows the *id* of the message. The sixth and seventh columns are used to insert items to the semaphores or queues to wake the specific tasks up. The difference between two columns is the sixth one wakes up a task to process the incoming data and the seventh one invokes a task to prepare data to transmit.

When the general features are defined, four time trigger matrices will be constructed in subsequently.

- *SEN-uC*

```
const unsigned char Trigger[TIMEMARKS][TRIGGER_INDEX] PROGMEM =
{
    {0, 0, 1, 't', 0x00, NONE, PR_DATA},    // Prepare the Sensor Data
    {1, 0, 1, 's', 0x11, NONE, NONE},       // Send Angles
```

```
{2, 0, 1, 'a', 0x00, NONE, NONE},
{3, 0, 1, 'a', 0x00, NONE, NONE},
{4, 0, 1, 'a', 0x00, NONE, NONE},
{5, 0, 1, 'a', 0x00, NONE, NONE}
};
```

PROGMEN: indicate that the time trigger matrix will be stored in flash memory, make more room for SRAM.

TIMEMARKS: is defined as “6”.

TRIGGER_INDEX: is defined as “7”.

Using **TIMEMARKS** and **TRIGGER_INDEX** is a convenient way to manage the time trigger matrices during coding phase.

This microcontroller gets sensor data from the sensors, save them on Sensor message then transmits them to other microcontroller. The detail of this matrix was explained below.

- Time slot 0: sends a semaphore to active Data_Preparation task to prepare sensor data.
- Time slot 1: Transmit sensor data via CAN BUS
- From time slot 2 to time slot 5: do nothing
- *PUSB-uC*

```
const unsigned char Trigger[TIMEMARKS][TRIGGER_INDEX] PROGMEM =
{
    {0, 0, 1, 'a', 0x00, NONE,    NONE}, //
    {1, 0, 1, 'a', 0x00, NONE,    NONE}, //
    {2, 0, 1, 'r', 0x11, RX_ANGLE, NONE}, // Receive Angles
    {3, 0, 1, 's', 0x12, NONE,    NONE}, // Send PUSB_uC
    {4, 0, 1, 'a', 0x00, NONE,    NONE}, //
    {5, 0, 1, 'a', 0x00, NONE,    NONE}
};
```

The PUSB-uC calculates the PWM from *Sensor* message and transmits to LCD-uC and MOTO-uC. The action of TTCAN Scheduler is explained following:

- Time slot 0: do nothing
- Time slot 1: do nothing
- Time slot 2: receives Sensor message
- Time slot 3: Send Pusb_uC message
- Time slot 4 and 5: do nothing
- *MOTO-uC*

```
const unsigned char Trigger[TIMEMARKS][TRIGGER_INDEX] PROGMEM =
{
    {0, 0, 1, 'a', 0x00, NONE,    NONE}, //
    {1, 0, 1, 'a', 0x00, NONE,    NONE}, //
    {2, 0, 1, 'r', 0x11, RX_ANGLE, NONE}, // receive Angles
    {3, 0, 1, 'a', 0x00, NONE,    NONE}, //
    {4, 0, 1, 'r', 0x12, RX_PUSB,  NONE}, // receive Pusb_uC
};
```

```
{5, 0, 1, 'a', 0x00, NONE, NONE}
};
```

MOTO-uC receives the *Sensor* and *Push_uC* messages then uses those data to control DC – motor.

- Time slot 0: receive reference message and manipulate the TCNT1 and OCR1A
- Time slot 1: do nothing
- Time slot 2: Receive Sensor/Angle message from sensor microcontroller.
- Time slot 3: do nothing
- Time slot 4: receive Push_uC message from Push microcontroller to adjust PWM
- Time slot 5: do nothing

- *LCD-uC*

```
const unsigned char Trigger[TIMEMARKS][TRIGGER_INDEX] PROGMEM =
{
    {0, 0, 1, 'a', 0x00, NONE, NONE}, //
    {1, 0, 1, 'a', 0x00, NONE, NONE}, //
    {2, 0, 1, 'r', 0x11, RX_ANGLE, NONE}, // Receive Angles
    {3, 0, 1, 'a', 0x00, NONE, NONE}, //
    {4, 0, 1, 'r', 0x12, RX_PUSB, NONE}, // receive PUSB_uC
    {5, 0, 1, 'a', 0x00, NONE, NONE}
};
```

The LCD-uC receives data from SEN-uC and PUSB-uC then shows the data on the LCD.

- Time slot 0: do nothing
- Time slot 1: Do nothing
- Time slot 2: Receive angle message from sensor microcontroller
- Time slot 3: do nothing
- Time slot 4: receives Push_uC message and displays the involved information on LCD.
- Time slot 5: do nothing

The CAN communication foundations are constructed. The next section introduces the TTCAN_Scheduler Task and CAN ISR design; those manage the basic cycle and correct the time drift of other microcontroller to the time master.

3.2.3 TTCAN_Scheduler Task and CAN ISR

This section describes the duties of TTCAN_Scheduler task and CAN ISR. Those functions are responsible for time synchronization, time cycle, and time slot duration in four microcontrollers therefore their structures should be analogous in every microcontroller. TTCAN_Scheduler task would manage the time cycle, the

time slot durations and when to transmit or receive a message. Meanwhile CAN ISR would manage the time synchronization.

- **TTCAN_Scheduler task**

As designed from the previous, in order to hold the pendulum at upright position, the time cycle is short, in 3 milliseconds. Every time window is equal 0.5 millisecond, which is also called time slot. The time slot runs from 0 to 5 then repeats. In each time slot, there is a specific action such as reception or transmission. Because of the role in the system, TTCAN_Scheduler tasks have different actions in a specific time slot.

On time slot 0, the master time microcontroller which is assigned for LCD-uC transmits a reference message. In the next time slot, the SEN-uC sends the Sensor message then in time slot 2 other microcontroller will receive the Sensor message. In the third time slot the PUSB-uC transmits the Push_uC message then this message is received by MOTO-uC and LCD-uC in the time slot adjacent latter. On the time slot 5 there is no action at all system.

- **CAN ISR**

CAN Interrupt Service Routine (ISR) happens if there is a message transmission or reception, or some error on the CAN Bus. In this content, it is assuming that there is no error on the communication. If CAN ISR happens, it means a message is sent or received. If a message is sent successfully, CAN ISR will reset the CAN control Data MOB register to wait for the next transmission. In the second case, when a message is received without error, depending on its identifier the message is stored on the respective *struct variable* such as *Angles* or *Push_uC*. If the message is a reference message, some the incoming data is used to correct the time drift of the received microcontroller with time master microcontroller.

In each node, it has its own time so-called local time. In order to perform TTCAN protocol, the local times in each node must be synchronized with the local time at master time node, which is now called global time. Therefore, as a basic cycle completed, the time master node transmits the reference message, which contains the information of the sending time point of the last reference message. The sending time point is saved on a variable so-called Ref_Time. When the reference message arrive a node, the receiving time of reference message in that node is saved on a variable which is called Rx_Time. Both the time point of last receiving time and sending point of reference message are also saved on two variables Last_Ref and Last_Rx. The time drift parameter is calculated by formula:

$$Time_drift = \frac{Rf_Time - Last_Rf}{Rx_Time - Last_Rx} \quad (3.1)$$

Then the local interrupt time of OCR1A register is manipulated:

$$OCRIA = Time_drift \times Compare \quad (3.2)$$

Where *Compare* is nominated interrupt time value for make a clock tick of FreeRTOS.

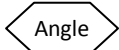

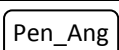
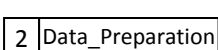
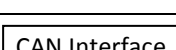
As the TTCAN_Scheduler task and CAN ISR design are completed for four microcontrollers, the next section will discuss about the specific tasks in each microcontroller which process the data to control the DC-motor.

3.2.4 Microcontroller Tasks

As mentioned in the former section, each microcontroller has assigned specific duties to make IRP working. SEN-uC detects the two angles changing, saves the change to global variables then transmits to other microcontrollers. LCD-uC receives data from other microcontroller and displays the information on the LCD. PUSB-uC receives Sensor message, calculates the PWM then transmits to MOTO-uC. The last microcontroller, MOTO-uC receives two message frames, *Sensor* frame and *Push_uC* frame. Based on the Pendulum angle, which derives from sensor frame, the MOTO-uC chooses which task would be invoked. After that, the chosen task manipulates the PWM then control the DC-motor.

In each microcontroller, there are several tasks working together. In order to choose which task would run first, which one run work second, it is necessary to create a prioritized preemptive scheduling [Richard Barry, 2004]. The priorities system is synchronized in all four microcontrollers. There are four priorities (0, 1, 2, 3), therein priority 0 is the lowest one and priority 3 is the highest one. The number of priority is configured in the header file *FreeRTOSConfig.h*. The lowest priority 0 is intended for idler task, the highest priority 3 is for *TTCAN_Scheduler* task in all 4 microcontrollers. Other priorities of each task in each microcontroller would be described in the respective section.

The tasks structures in microcontrollers would introduce in the following section. Before going deeply to the detail, the table below shows several symbols and their meaning which are used in latter section.

	Message frame
	Queue or semaphore
	Global variable
	Task or ISR (the first index is priority, the second is the task name)
	CAN Interface

- **SEN – uC**

There are two duties in this microcontroller. The first one is to detect the changing of two angles: Pendulum and Arm angle then store the new values to the respective global variables. The second one is to prepare the Sensor data which derives from two global variables in time slot “0” then transmit to other microcontroller in the time slot “1”. In order to fulfill the first duties, as presented in the section 2.2.3, external ISRs are applied here. Because of the bidirectional of two angles, four external ISR are deployed. The two global variables are named *Pen_Ang* and *Arm_Ang*.

In the time slot “0” of every basic cycle, global variables *Pen_Ang*, *Arm_Ang* are saved to four data bytes of *Sensor/Angle* message, which is introduced in section 3.2.2. In order to transfer the values of *Pen_Ang* and *Arm_Ang* to *Angle* message, a task would be built to do that. The task is named *Data_Preparation*. It is invoked every three millisecond by semaphore signal [Richard Barry, 2004], which is controlled by *TTCAN_Scheduler* task. Since the *Angle* message is ready, it waits for being transmitted by *TTCAN_Scheduler*. When the time slot “1” comes, *Angles* message is transmitted to other microcontroller via CAN bus. The whole process is presented in the figure below.

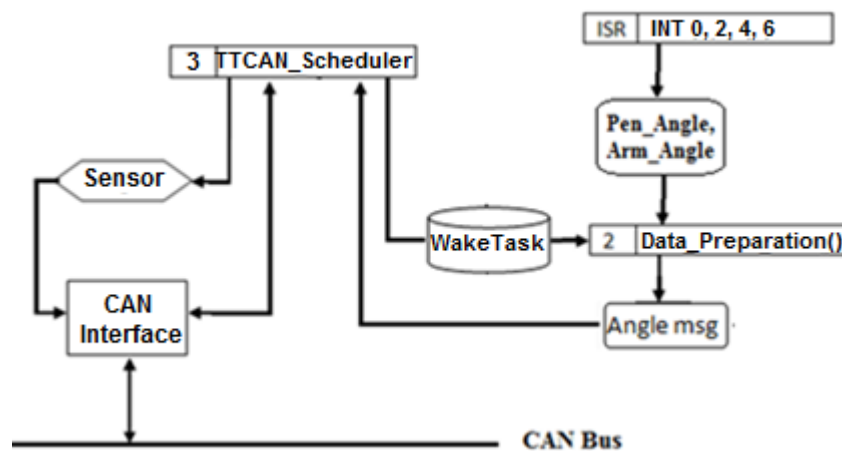


Figure 3.10 Time Trigger Plan of SEN-uC

When the IRP moves, its two angles change values. The changing is detected by external ISR and they save the new values of two angles to two respective global variables. When the time slot “0” comes, *TTCAN_Scheduler* task wakes the task *Data_Reception()* up by semaphore *WakeTaskSe*. The task saves the new values of *Pen_Angle* and *Arm_Ang* to *Angle* message. In the next time slot, *TTCAN_Scheduler* will send the *Angle* message.

There are only two tasks in SEN-uC. The first one is “*Data_Preparation*”, which was described earlier. The second one is “*uC_Check*”, which is used to check the microcontroller state. Because of the task’s roles, the task *Data_Preparation()* is given priority “2”, the residual one has priority “1”.

- **LCD – uC**

This microcontroller receives two messages *Sensor* and *Push_uC* then display the needed data on the LCD. Because the duties are separate, two tasks will be built to perform them. The first task which receives the two messages is named *Data_Reception()*. The second one which is named *LCD_Display()* displays the desired information on the LCD. Because of receiving of two messages, *struct variables* are created to deal with two messages. There are also two global integer variables to store the angles. The task and *TTCAN_Scheduler* are shown on the figure below.

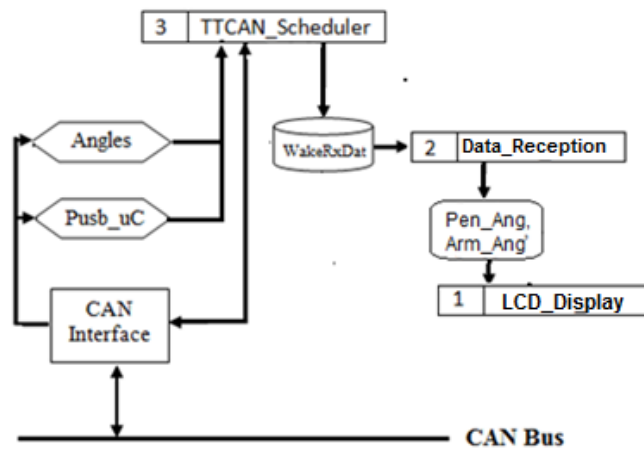


Figure 3.11 Time Trigger Plan of LCD-uC

When a message comes on a specific time slot, the *TTCAN_Scheduler* task sends an item on the queue to wake up the *Data_Reception()* task. Then the task processes the data from the message, change the two global variables, *Pen_Ang* and *Arm_Ang*. The two angles values are shown on the LCD by the *LCD_Display()* task. The *Data_Reception()* is given the priority “2” and the “1” priority is for *LCD_Display()* tasks.

On the LCD, some information should be shown:

- The greeting to the program: *Welcome to IRP*
- The states of IRP: *(Power on, Change input, swing up, Steady State, Stop)*
- Two angles of IRP: *P_Ang, A_Ang*
- PWM:
- Reference input: *Ref. Input*
- Started/Stopped Button: *display which to start, which to stop the IRP*
- Change the ref. input button: *display which button to increase, which button to decrease Ref. Input*
- Finish changing ref. Input button: *display which button finish the changing process*
- Button to start or stop oscillation

It is needed at least eleven rows on the LCD to show all the necessary information. Because of the limitation of the LCD, which can only display eight rows, the necessary information is shown on two screens. The first screen displays like the figure following:

Welcome to IRP
States: Power on/Swing up/Steady State/Stop
P_Ang: 180 deg.
A_Ang: 20 deg.
PWM: 70%
Ref. Input: 20 deg.
Push B1 to start/ Push B2 to stop
Push B3 to change the ref. input /

The second screen displays the following information:

States: Change Ref. Input
A_Ang: 30.04
Ref. Input: 20 deg.
Push B4 to increase ref. Input
Push B5 to decrease ref. Input
Push B6 to finish
Button 7: Oscillation
Button 8: Stop oscillation /

The structure design of LCD is completed. There are two implemented tasks in this microcontroller, they are *Data_Reception()* and *LCD_Display()*. All the information of the IRP is shown on two screens, which were described above. The next part will discuss about the MOTO-uC.

- **PUSB– uC**

This microcontroller receives the *Angles/Sensor* message on time slot 2; processes the data on that message then store the result on *Pushb_uC* message. Moreover, PUSB-uC detects the pushed buttons which control the operation of the IRP. If the buttons are pushed, they can change the global variable – *NomInput* – which is responsible for reference input or save a new value on a specific byte on the *Pushb_uC* message. On time slot 3, the *Pushb_uC* message is transmitted via CAN bus by *TTCAN_Scheduler* task. The two duties here are separated; they can be implemented by two tasks in parallel. The task which calculates the PWM is called *PWM_Calculation()*. It is trigger by Cycle Time of TTCAN Scheduler. The task which is responsible for the pushed buttons is named *PushedButton_Detection()*. The principal operation of this task is different to the first one. The second task operates basing on event trigger. If any button is pushed, the task will detect and

save the respective data or signal on the *Pusb_uC* message. The operation of microcontroller tasks is shown on the figure 3.11 below.

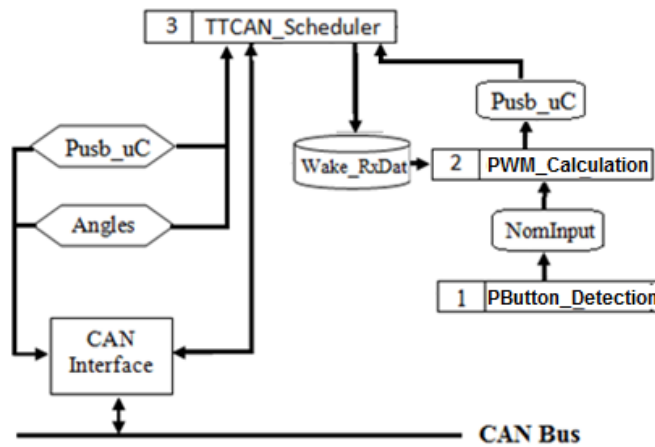


Figure 3.12 Time Trigger Plan of PUSB-uC

In the time slot 2, *TTCAN_Scheduler* task sends a semaphore to wake up the *PWM_Calculation()* to calculate PWM. The *PWM_Calculation()* task uses the data from *Sensor* message and *NomInput* in its process. The result is saved on the data bytes of *Pusb_uC*; the message then waits for *TTCAN_Scheduler()* task which sends the message on time slot 3. *NomInput* variable is only changed if the increased or decreased reference input buttons are pushed.

Because of the roles of each task and processed time, the priority “1” is for *PushedButton_Detection()* task, and *PWM_Calculation()* task’s priority is “2”.

As mentioned, the buttons control the operation of IRP. It can stop or start the system, increase or decrease the reference input or oscillate the arm. In the LCD-uC section, there are two screens which display the necessary information. In order to do that, it is needed two more buttons. One is for switching from screen 1 to screen 2, and the other is for opposite direction. With that design, there are eight buttons applied. Because the pushed button matrix has 12 buttons, the used buttons are chosen as following:

- Button 1 to start the IRP operation
- Button 2 to stop the IRP operation
- Button 3 to switch to the second screen
- Button 4 to increase the reference input
- Button 5 to decrease the reference input
- Button 6 to apply the new reference input and return the first screen.
- Button 7 to start oscillation
- Button 8 to stop oscillation

- **MOTO – uC**

MOTO-uC receives two message *Angles* and *Pusb_uC* on time slot 2 and 4 respectively. The data on both messages are applied to control the IRP swinging up and stay at upright position. To perform those three jobs, three tasks are built to implement each job separately. The first task is named *Data_Reception()*, which processes with all the incoming CAN message. The second task which swings the pendulum up, is called *SwingUp_Controller()*. The third one is *Upright_Controller()*. This task's duty is to hold the pendulum at upright position.

Each time a message comes, *TTCAN_Scheduler* task give a semaphore which is called *Wake_RxData* to wake the *Data_Reception* task up. From the data on the *Sensor/Angles* message, the *Data_Reception()* task calculates the current Pendulum Angle of the IRP. If the Pendulum angle is between (-165; 165) degree, a semaphore – *Wake_SwingUp* – is sent to wake up the *SwingUp_Controller()* task. Otherwise, *SwingUp_Controller()* is invoked by another semaphore which is named *Wake_Upright*. Both the *SwingUp_Controller()* and *Upright_Controller()* tasks can change the register OCR3B, which manipulates the PWM and therefore controls the DC-motor.

On the *SwingUp_Controller()* task, the PWM is fixed and the energy is supplied to the DC-motor as a constant amount in every period. On the other side, the PWM on the *Upright_Controller()* task derive from *Pusb_uC* message. It varies to supply appropriate energy to hold the IRP at upright position.

If there is a controlled signal from the stopped button of PUSB-uC, the PWM is set to zero. The working schedule of the MOTO-uC is shown on the figure below.

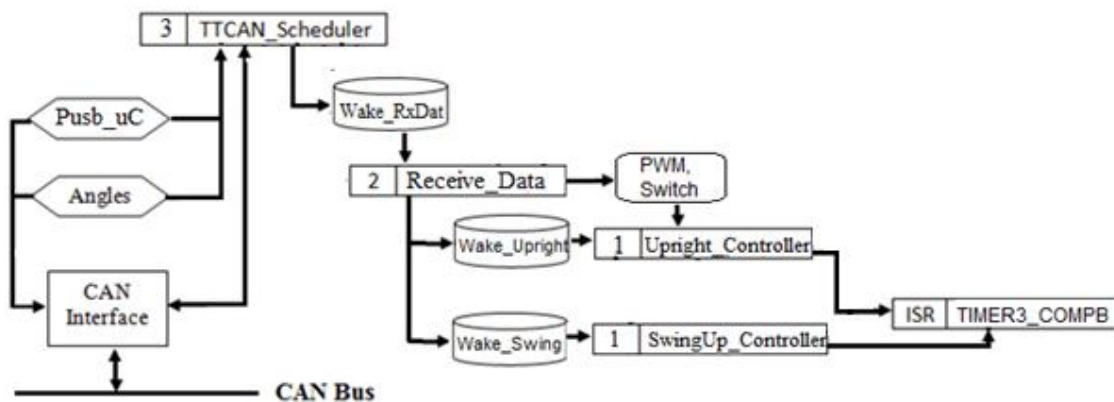


Figure 3.13 Time Trigger Plan on MOTO-uC

Similar to other microcontroller, MOTO-uC also has two messages, *Pusb_uC* and *Angles*. When a message arrives, a semaphore is sent to *Data_Reception()* task. It wakes up and calculates the Pendulum angle. Depend on the pendulum angle a semaphore is sent to wake one of two tasks *Upright_Controller()* or

SwingUp_Controller(). The second one set PWM equal to a predefined value. On the other hand, the first task is set PWM equal to the data which is from *Pusb_uC* message.

In three task, *Data_Reception()* task wake two others tasks, so it has the priority “2”. The residual tasks are set to priority “1”.

4 Implementation

From chapter 3, the microcontroller structures are organized. Base on the knowledge, this chapter would be developed the work. This chapter consists of seven sections. The first section introduces the implementation of the nonlinear IRP model. From the section 4.2 to the section 4.6, the control program codes are described in detail. Then the last section gives several directives when estimating the system parameters. After this chapter, readers are able to understand how the IRP operation and by what precise code. However, because of the program volume, the main parts of the code are described in this section. The entire programs are presented on the Appendix part and the attached CD-ROM.

4.1 Nonlinear IRP model

There are two models of the IRP which are corresponding to the minimal and maximum goals of the master thesis. In the first IRP model, the reference input is always zero, it means the IRP operates at only position (both Pendulum and Arm angles are zero). The second IRP model has nonzero reference input therefore the system can work at any arbitrary arm angle. In order to implement that, there is one more state added to the state vector, and the IRP has 5 states to control DC-motor.

The method which is used to find the feedback signal is Linear Quadratic Regulator (LQR) [Bui, 2013]. The LQR method is support by the function *lqr()* in the Matlab program. Two weighting matrices Q and R are the input of the function and the output are the feedback parameters. Q and R , ρ represent the relation between the response of system and control effort. They are related by the formula:

$$J = \int_0^{\infty} [\underline{x}^T(t) \underline{Q} \underline{x}(t) + \rho \underline{u}^T(t) R \underline{u}(t)] dt$$

- **Feedback for the nonlinear IRP model with zero reference input**

In order to operate in the linear range, the working range of pendulum is small about 0.2 radian, therefore $Q_{(2,2)}$ can be chosen equal to 30. The arm angle can work in larger range without affection to the IRP stability, so the $Q_{(1,1)}$ is chosen equal to 4. In addition, the pendulum and arm angular velocity $\dot{\phi}$ are almost not limited, therefore the $Q_{(4,4)}$ and $Q_{(3,3)}$ are chosen equal to 0.5. The matrix \underline{Q}_1 is shown as following:

$$\underline{\underline{Q_1}} = \begin{bmatrix} 4 & 0 & 0 & 0 \\ 0 & 30 & 0 & 0 \\ 0 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 0.5 \end{bmatrix};$$

The matrix R in this situation is just a value because there is only one input. The control penalty R and ρ are determined by trial and error. First, it is chosen:

$$R = 0.1;$$

$$\rho = 10;$$

Input all the IRP matrices:

$$\underline{\underline{A_1}} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & P_1 & P_3 & 0 \\ 0 & P_2 & P_4 & 0 \end{bmatrix}; \quad \underline{\underline{B_1}} = \begin{bmatrix} 0 \\ 0 \\ P_5 \\ P_6 \end{bmatrix}; \quad \underline{\underline{C}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}; \quad \underline{\underline{D}} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

The `lqr()` function is executed by command:

```
% find the control law based on the LQR
K1 = lqr(A1, B1, Q1, rho*R)
```

The feedback regulator gains were obtained:

$$K1 = [-2.707 \quad 51.563 \quad -5.369 \quad 9.934],$$

- **Feedback for nonzero reference input model**

To deal with the nonzero reference input, an error state is added to the IRP system [Bui, 2013]. From now on, the IRP has five states and all of them involves to the control process. Most the system matrices increase one dimension as following:

$$\underline{\underline{Q_2}} = \begin{bmatrix} 4 & 0 & 0 & 0 & 0 \\ 0 & 30 & 0 & 0 & 0 \\ 0 & 0 & 0.2 & 0 & 0 \\ 0 & 0 & 0 & 0.2 & 0 \\ 0 & 0 & 0 & 0 & 0.5 \end{bmatrix}; \quad \underline{\underline{A_2}} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & P_1 & P_3 & 0 & 0 \\ 0 & P_2 & P_4 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}; \quad \underline{\underline{B_2}} = \begin{bmatrix} 0 \\ 0 \\ P_5 \\ P_6 \\ 0 \end{bmatrix}$$

Similarly, the procedure to calculate the feedback parameter is repeated:

```
% find the control law based on the LQR
K = lqr(A2, B2, Q2, rho*R)
```

The feedback regulator gains were obtained:

$$K2 = [-3.926 \quad 55.348 \quad -5.863 \quad 10.665 \quad -1.414],$$

The two parameter matrices are then applied to the respective Simulink model and the PWM process to control DC-motor.

4.2 The common parts

There are some common parts in all four microcontrollers such as the Can ISR, CAN initialization, reset the received MOB, Transmitted MOB, and TTCAN scheduler. To avoid the repeating, all those parts will be introduced in this section. Firstly, the FreeRTOS tick clock is set 0.5 milliseconds by assigning the configTICK_RATE_HZ equal to 2000 and the total priority level in the whole microcontroller is 4.

```
#define configTICK_RATE_HZ      ( ( portTickType ) 2000 )  
#define configMAX_PRIORITIES   ( ( unsigned portBASE_TYPE ) 4 )
```

Then the prescaler of FreeRTOS clock – Timer/Counter1 is set to 8 to synchronize with the smallest prescaler of CAN Timer. When the Timer/Counter1 prescaler is 8, the TCNT1 counts from 0 to 999 (equal 0.5 milliseconds). The values in the table below would be set to the respective register to reach the design requirements. All the define value here belong to the *ConfigFreeRTOS.h* and *port.c*.

```
#define portCLEAR_COUNTER_ON_MATCH ( ( unsigned portCHAR ) 0x08 )  
#define portPRESCALE_8           ( ( unsigned portCHAR ) 0x02 )  
#define portCLOCK_PRESCALER      ( ( unsigned portLONG ) 8 )  
#define portCOMPARE_MATCH_A_INTERRUPT_ENABLE ( ( unsigned portCHAR ) 0x02 )
```

Then the next section introduces the CAN Tasks.

4.2.1 CAN Tasks

All the tasks which control the CAN controller operation belong to the TTCAN.c and TTCAN.h files. In the TTCAN.h some defines and variables are declared, meanwhile the TTCAN.c contains the task code.

- **TTCAN.h**

In this header file, the parameters which are controlled the TTCAN scheduler are declared. They are Basic Cycle duration *BASIC_CYCLE*, the number of basic cycle *CYCLE_MAX*, the number of time marks (time slots) *TIMEMARKS*, the number of column in the trigger matrix *TRIGGER_INDEX*, the baud rate of transmission *BAUDRATE*, the nominal value of OCR1A *COMPARE*, ordinal number of the MOB which stores the incoming or sending message *R_MOB* and *S_MOB* and the masking ID of the received message. All the parameters above are the same for all four microcontrollers, which are shown on the figure below. The only different parameter is the Time Master *TM*. If the microcontroller is Time Master, *TM* has value 1.

Otherwise the value “0” is assigned to the *TM*. Two struct variables are declared with the names *Angles* and *Pusb_uC* which store the content of CAN message.

```
#ifndef TTCAN_H_
#define TTCAN_H_
#define BASIC_CYCLE      6    //Duration of Basis cycle (ms)
#define MAX_CYCLE        1    //number of Basis cycle (scope of System matrix)
#define TIMEMARKS        6    //number of time mark (= row number of trigger matrix)
#define TRIGGER_INDEX    7    //number of index in Trigger matrix
#define TM                0    // = 1: Time master. = 0 not Timer master
//set up Baud rate (_100KBPS, _125KBPS, _200KBPS, _250KBPS, _500KBPS, _1MBPS)
#define BAUDRATE _500KBPS
#define COMPARE          999   //compared value of FreeRTOS-Timers
#define R_MOB            1     //Number of receipt-MOB
#define S_MOB            0     //Number of Sending-MOB
#define ID               0x00  //receipt ID for Masking
#define MSK              0x00  //receipt masking
//Baud rate
#define _100KBPS         0
#define _125KBPS         1
#define _200KBPS         2
#define _250KBPS         3
#define _500KBPS         4
#define _1MBPS           5
#define TRUE             1
#define FALSE            0
#endif BAUDRATE
#error Keine Baudrate festgelegt!
#endif
struct CAN_message
{
    unsigned int id;        //MSG ID 11 Bit
    unsigned char length;   //DLC Length
    unsigned char data[8];  //Data field 8 Byte
} Angle, Pusb_uC;
void TTCAN_Init();
#endif /* TTCAN_H_ */
```

*** CAN Initialization – CAN_Init(unsigned char bitrate)**

In this function all the registers of CAN controller and 15 MOBs are configured. Firstly, all the registers which control in general the CAN operation are reset. The list of registers is reset as following:

```
CANGCON = 0x00;           //Reset the General Control Registers
CANGCON |= 0b00000001;    //Reset enabled for the CAN-Controllers (SWRES bit)
CANGIT = 0x00;           // Register for interrupt flag
CANGIE = 0x00;           // Register for Enable interrupt
CANSIT2 = 0x00;          //Register for Status of interrupt by MOB
CANSIT1 = 0x00;
CANEN1 = 0x00;           // Register for check MOB availability
CANEN2 = 0x00;           // 1 = in use. 0 = empty (can receive data)
CANIE1 = 0x00;           // Register for Enable MOB Interrupt
CANIE2 = 0x00;
CANTCON = 0x00;          // prescaler = 8*(CANTCON + 1)
```

The CANTCON register set the prescaler of CAN timer equal to “1”, the CAN timer counts with the scale of 8 to the CPU clock.

The next step is reset all the MOB registers. In order to reset 15 MOBs, a **for**-loop is applied to run from the MOB “0” to MOB “14”. The state of MOB in the reset process is deactivated.

```
for (mob = 0; mob < 15; mob++)    //All MOBs is set Null
{
    CANPAGE = (mob << 4);          // Go to the next MOB
    CANIDT1 = 0x00;                // Reset CAN Identifier Tag Registers
    CANIDT2 = 0x00;                // used to set kind of message (remote, data)
    CANIDT3 = 0x00;
    CANIDT4 = 0x00;
    CANIDM1 = 0x00;                // Reset CAN Identifier Mask Registers
    CANIDM2 = 0x00;                // used to comparison true forced (filter)
    CANIDM3 = 0x00;
    CANIDM4 = 0x00;
    CANSTMOB = 0x00;              // Reset MOB-Status
    CANCDMOB = 0x00;              // Deactivate MOB
}
```

After resetting, the registers are configured the new desire working condition. The configured parameters are: The baud rate of transmission, the direction of pins 5 and 6 of port D, CAN interrupts, global interrupt and TTC mode. The configuration detail is shown as following:

```
DDRD |= 0b10100000;              //Pin D5 output (TXCAN)
DDRD &= 0b10111111;              //Pin D6 input (RXCAN)
//Auto-Increment index bit3 = 0, when FIFO
CANPAGE &= 0b11110111;
//CAN Bit Timing Register
CANBT1 = pgm_read_byte(&baudrate[bitrate][0]); // Set Baud Rate Prescaler
CANBT2 = pgm_read_byte(&baudrate[bitrate][1]); //
CANBT3 = pgm_read_byte(&baudrate[bitrate][2]); //
CANGIE |= 0b10111000;            //ENIT-, ENRX-, ENTX-, ENERR-Interrupt activate
CANGCON |= 0b00100010;           //TTC mode activate - Enable CAN
while (!(CANGSTA & 0x04));        //wait until CAN-Controller ready
sei();
```

- **CAN message Reception** – *void CAN_RX()*

This function sets the condition to receive or reject an incoming CAN message. First the MOB which receives and stores the CAN message is identified by *R_MOB*. Then the interrupts which watches the received process are activated. If the reception is successful or any error happens, the respective interrupts will inform the CAN controller. The next step is to set the Identifier Tag and received filter. Identifier Tag decides which bits of the message Identifier are compared and received filter decides what values on the compared bits are. After that the control Data MOB register is configured to be ready receiving a CAN message. The code of the task *Can_RX()* is shown as following.

```

void Can_RX()
{
    CANPAGE = (R_MOB << 4);    //Select a MOB to store the coming frame
    #if R_MOB>7                //Enable the respective Interrupt for the MOB
        CANIE1 |= (1 << (R_MOB-8));
    #else
        CANIE2 |= (1 << R_MOB);
    #endif
    //Set Identifier Tag (allow to receive all frame)
    CANIDT1 = (unsigned char) (ID>>3);
    CANIDT2 = (unsigned char) ((ID<<5) & 0xE0);
    //Set received filter (no filtering)
    CANIDM1 = (unsigned char) (MSK>>3);
    CANIDM2 = (unsigned char) ((MSK<<5) & 0xE0);
    CANCDMOB &= 0b10111111;    // set bit6 = 0
    CANCDMOB |= 0b10000000;    // Enable reception
}

```

- **CAN message Transmission** – *void CAN_TX(struct CAN_message msg)*

This function configures the condition to transmit a CAN message. First the MOB which stores the transmitted CAN message is identified by *S_MOB*. Then the interrupts which watches the received process are activated. If the reception is successful or any error happens, the respective interrupts will inform the CAN controller. The next step is to set the data length and the Identifier of the message. Then the data is saved on the CANMSG registers. After that the Control Data MOB register is configured to be ready sending the CAN message. The code of the task *Can_TX()* is shown as following.

```

void Can_TX(struct CAN_message msg)
{
    unsigned char i;
    CANPAGE = (S_MOB << 4);    //select MOB storing Data to be sent
    #if S_MOB>7                //Enable respective Interrupt for the MOB
        CANIE1 |= (1 << (S_MOB-8));
    #else
        CANIE2 |= (1 << S_MOB);
    #endif
    CANCDMOB &= 0b11100000;    //reset CAN V2.0A and reset DLC = 0
    CANCDMOB |= msg.length;    //set the length of Data field
    // Get Identifier of the up-sending frame
    CANIDT1 = (unsigned char) (msg.id>>3);
    CANIDT2 = (unsigned char) ((msg.id<<5) & 0xE0);
    //Write Data on CANMSG (Pointer with Auto-Increment)
    for (i=0; i<msg.length; i++) CANMSG = msg.data[i];
    while (CANGSTA & 0x10);    //Wait until CAN Transmitter available
    CANCDMOB &= 0b01111111;    // set bit7 = 0
    CANCDMOB |= 0b01000000;    // Enable transmission
}

```

- **Detect operating MOB** – *unsigned char Get_Operating_MOB (unsigned int Register)*

When a message is sent or receiving successfully, the CAN controller is informed. Then, the CAN controller identifies which MOB hosts the activity. The finding process is implemented by the task *Get_Operating-*

_MOB(). The input of the task is the CANSIT register. If an interrupt happens by a specific MOB, the respective bit on this register is set to “1”. This task checks every bit of the register then returns the operating MOB number. The code of the function is shown below.

```
unsigned char Get_Operating_MOB(unsigned int Register)
{
    unsigned char mob_number;
    if (Register == 0) return 0xFF;
    // check from the first to the last MOB
    for (mob_number = 0; (Register & 0x01) == 0; ++mob_number) Register >>=1;
    if (mob_number > 14) return (0xFF);
    else return mob_number;
}
```

- **TTCAN Scheduler** – *TTCAN_Scheduler(void *pvParameters)*

This task manages the CAN controller operation. Every activity of the CAN controller such as message transmission, reception, time slot duration, basic cycle, time marks are controlled to following the predefined design. In this function, there are some variables which should store a value and is used for the next calculation. In order to ensure the data is not lost, the variables are declared as static.

```
struct CAN_message ref_message;
static unsigned char Cycle_Number=0;
unsigned char time_mark, base_mark, repeat, type, queue = 0;
portTickType xLastWakeTime;
```

The *ref_message* variable is to deal with the reference message. *xLastWakeTime* is the last wake up time point of this function. Every 0.5 millisecond the task is waked up based on this time point. Other variables are used to control the action every basic cycle.

The first command in this task is to get the time of local time. Then the task goes to a while-loop, which runs forever. The *Cycle_Time* is the variable which measure the time elapsing in one basic cycle. If the *Cycle_Time* is zero, and the microcontroller is master time, a reference message is created and sent to other microcontroller.

```
if (Cycle_Time == 0 && TM == 1) //At time mark zero and is Time master
{
    // Create RM to adjust the local time to global time
    ref_message.id=0x00; //RM's ID
    ref_message.length=2;
    ref_message.data[0]=(char) (TXtime >> 8); //Send the last transmitted time
    ref_message.data[1]=(char) TXtime;
    Can_TX(ref_message);
}
```

The next command is to read the *time_mark*, *base_mark*, *repeat* and *type* variables from the trigger matrix at a specific row.

```
time_mark = pgm_read_byte (&Trigger[Cycle_Time][0]);
base_mark = pgm_read_byte (&Trigger[Cycle_Time][1]);
repeat = pgm_read_byte (&Trigger[Cycle_Time][2]);
type = pgm_read_byte (&Trigger[Cycle_Time][3]);
```


If the *time_mark* value is equal to *Cycle_Time* and the *base_mark* and *repeat* are appropriate, the task will choose the appropriate action, which is done by *type* variable.

```
if (Cycle_Time == time_mark)           //Reached the time mark?
{
    //Check whether the frame provided in this cycle?
    if (((Cycle_Number-base_mark)%repeat==0 && Cycle_Number>base_mark)||Cycle_Number == base_mark)
    {
        switch (type)
        {
            case 'R':
            case 'r':    queue = pgm_read_byte (&Trigger[Cycle_Time][5]);    //receive message
                        if (queue != 0) xQueueSend(Wake_RxData, &queue, (portTickType) 0); break;
            case 'S':
            case 's':    //send message
            case 'A':
            case 'a':
            case 'T':
            case 't':
            default: break;
        }
    }
}
```

If the *type* variable is equal to 'R' or 'r', the CAN controller will receive a CAN message, therefore a semaphore or queue is sent to wake up a task, which processes the data on the CAN message. In the figure above, a value is sent to the queue *Wake_RxData* to wake the *Data_Reception()* task up. If the *type* has value such as 'S' or 's', a CAN message will be transmitted. If the *type* is 'A' or 'a', that time slot is for the arbitrary slot, which will be used in future. If the *type* is 'T' or 't', a specific task will be woke up to prepare the data. The actions in each time slot are difference depending on microcontroller's duties.

The next code part of this task is to check the *Cycle_Time* and *Cycle_Number*. Because *Cycle_Time* is a limited time for a basic cycle; therefore if it is over the basic cycle time, it is reset to "0". Otherwise it is added one more value. After that, the *TTCAN_Scheduler()* task waits 0.5 millisecond for the next time slot. The respective code is shown as following:

```
if (Cycle_Time == BASIC_CYCLE-1)    //If the end of the basic cycle, cycle time reaches to zero
{
    Cycle_Time = 0;
    if (Cycle_Number < MAX_CYCLE-1) Cycle_Number++;
    else Cycle_Number=0;             //Another cycle or the end of the system matrix achieved?
}
else Cycle_Time++;
vTaskDelayUntil(&xLastWakeTime, 1);    // wait 0.5ms
```

4.2.2 CAN ISR

CAN ISR's duties is to reset CAN Control Data MOB register when a message is sent or received successfully. Moreover, when a reference message comes, the local time synchronization is implemented in this ISR. In order to synchronize the local time in each microcontroller, some variables are declared as static,

which can store the value despite of the ISR operation complement. The variables which are used in the CAN ISR are shown in the table below.

```
struct CAN_message buffer;
unsigned char save_canpage, mob, i=1;
static unsigned char sync = FALSE;
static unsigned int last_ref, last_rx;
static unsigned long int locdiff = 999*BASIC_CYCLE;
```

There is a struct CAN_message variable *buffer* which is used to temporary to store the data from a specific MOB in CAN ISR. There are some static variables such as *last_ref*, *last_rx*, *locdiff* which are used to correct the time drift between a microcontroller and the time master.

When an interrupt happens, the ISR will find where the Interrupt comes from, and jump to the found MOB. Then the Identifier of message which is stored in the MOB is copied to *buffer.id*.

```
save_canpage = CANPAGE;           //insure CANPAGE
mob = Get_Operating_MOB(CANSIT); //Get the MOB which its Interrupt is running?
if(mob==0xFF) return;
CANPAGE = (mob<<4);               //Go to the MOB
buffer.id = (unsigned int) (CANIDT1<<8); // Determine ID
buffer.id |= (CANIDT2 & 0xE0);
buffer.id >>= 5;
```

The next action of this ISR is to check the CANSTMOB register. If the MOB has just sent a message successfully, there are some action should be done with. If the sent message is reference one, the transmitted time point is saved on a global variable Txtime. This variable is also used in the *TTCAN_Scheduler()* task to create a reference message. Then the CANCDMOB register disable the transmitted state of the MOB to wait the next sending message. The respective code is shown below.

```
if (CANSTMOB & 0b01000000)           //Sending was successful?
{
    if (buffer.id == 0x00) Txtime = CANTTC; // Get transmitted time of RM
    CANSTMOB &= 0b10111111;             //Delete Transmit-Flag
    CANCDMOB &= 0b10111111;             //disable Sending
}
```

If the CANSTMOB informs that the MOB has just received successful a CAN message, the data and the length of the message are transferred to the *buffer* variable. If the received message is reference message and the received microcontroller is not master time, the message's data is used to correct the time drift with the Time master microcontroller. If the received message is an ordinary one, basing on its identifier the data is saved on the respective *struct CAN_message* variable such as *Angles* or *Pusb_uC*. The detail of the code is shown below.

4.3 SEN-uC

There are two duties which are implemented in this microcontroller. The first one is to use external ISR to detect then change of Pendulum angle Arm angle then change the respective global variables. In order to fulfill this duty, as presented in the last chapter, there are four external ISR deployed to detect the change of Pendulum angle and Arm angle. The second one is to prepare the sensor data which derive from two global variables in time slot 0. The Task *Data_Preparation()* will implement the second duties. The two global variables which store the two angle is called *Pendulum_axis* and *Arm_axis*.

4.3.1 Specific External Interrupt Service Routines

The four external ISRs are used which are external ISR 0, 2, 4 and 6. The zero and second external interrupt are used to detect the change of the first sensor, which is in charge of global variable *Pendulum_axis*. The fourth and the sixth external interrupts are responsible to sense the signal from the second sensor, which manipulate the variable *Arm_axis*. If there is a change on one track of the encoder, the respective pins on the SEN-uC will jump the value from “0” to “1” or vice versa. There are two tracks which are watched on each encoder. There are 500 windows on each track; combining of two tracks the encoder can distinguish 2000 positions per one revolution. The code inside the External ISR is shown in the figure below.

```
ISR(INT0_vect)
{
    // Pin A of quadrature encoder
    if (((PIND & 5) == 4) || ((PIND & 5) == 1))
        Pendulum_axis++;
    else Pendulum_axis--;
}
//-----
ISR(INT2_vect)
{
    // Pin B of quadrature encoder
    if ( ((PIND & 5) == 0) || ((PIND & 5) == 5) )
        Pendulum_axis++;
    else Pendulum_axis--;
}
//-----
ISR(INT4_vect)
{
    // Pin A of quadrature encoder
    if ( ((PINE & 80) == 64) || ((PINE & 80) == 16) )
        Arm_axis++;
    else Arm_axis--;
}
//-----
ISR(INT6_vect)
{
    // Pin B of quadrature encoder
    if ( ((PINE & 80) == 0) || ((PINE & 80) == 80) )
        Arm_axis++;
    else Arm_axis--;
}
```

The external ISR “0” and “2” watch the track 1 and track 2 respectively on the first encoder. If there is any movement of the first encoder, the two pins 1 and 3 of port D are checked. As external ISR 0 detects the

change, and the values of pin 1 and 3 are opposite it means Pendulum angle increases. If two pin values are same sign, the angle decreases the value. The second external ISR operates in a different way. When this ISR senses the change of the second track, if values of pin 1 and 3 are same sign, *Pendulum_axis* variables is added more one value. If not, *Pendulum_axis* declines one. The fourth and sixth external ISRs work with the same principal to vary the *Arm_axis* variable.

4.3.2 Data_Preparation() Task

This task prepares the sensor data for the *Angles* message. It is invoked by semaphore which is sent by TTCAN_Scheduler at time slot 0 in every basic cycle. In this task, the *Pendulum_axis* and *Arm_axis* values are transferred to the struct variable *Angle*. If the values are over 1999 or lower than -1999, they are reset equal to the remainder of the division of the values and 2000. When the *Angle* struct variable is set, it waits for being transmitted.

```
void Data_Preparation(void * pvParameters)
{
    int Buffer1, Buffer2;
    DDRF = 0xff;
    PORTF = 0;
    for (;;) // Start of infinity loop
    {
        if (xSemaphoreTake(WakeTaskSe, portMAX_DELAY))
        {
            // Save pendulum angle
            if ((Pendulum_axis >= 2000) || (Pendulum_axis <= -2000))
            {
                Pendulum_axis = Pendulum_axis % 2000;
            }
            Buffer1 = Pendulum_axis;
            Pen_vel = Pendulum_axis - Last_Pendulum_axis;
            Angle.data[0] = (unsigned char)(Buffer1 >> 8);
            Angle.data[1] = (unsigned char)(Buffer1 & 0x00ff);
            // Save arm angle
            if ((Arm_axis >= 2000) || (Arm_axis <= -2000))
            {
                Arm_axis = Arm_axis % 2000;
            }
            Buffer2 = Arm_axis;
            Arm_vel = Arm_axis - Last_Arm_axis;
            Angle.data[2] = (unsigned char)(Buffer2 >> 8);
            Angle.data[3] = (unsigned char)(Buffer2 & 0x00ff);
        }
    } // end infinity loop
} // end of task Get_Sensor_Data
```

4.4 LCD-uC

LCD-uC controls the LCD screen, a part of the Human Machine Interface component. In order to display the information on the LCD, there are two tasks called *Data_Reception()* and *LCD_Display()*. The former task is to deal with the two incoming message. The latter one shows the received data on the LCD. The meanings of the codes are explained in this section. The entire code which is presented in order can be seen in the Appendix part or the attached CD-Rom.

- **Data_Reception() Task**

When a message comes and are saved on the respective struct variables, a value is sent by the *TTCAN_Scheduler* task to the *Wake_RxData* queue and wake the *Data_Reception()* up. In order to reduce the time process, depend on the incoming message, only the appropriate code of the task would run. First, several variables are declared to perform in this task. There are three integer variables *buffer1*, *buffer2*, *buffer3* which receive data from the messages and an unsigned char variable – *Check_Slot* to check what kind of message and give the appropriate response.

```
// Local variables
int buffer1, buffer2, buffer3;
unsigned char Check_Slot = 0;
```

As the *Angles* message comes, a value equal to *RX_ANGLE* (equal to 1) is sent to the *Wake_RxData* queue. This value is the condition for running the code which processes the data from the Angle message and saves those results on two variables *DPen_Angle* and *DArm_Angle*. The two angles are shown on the LCD latter.

```
if (Check_Slot == RX_ANGLE)    // if receiving a angle message
{
    // receive data from the AngleQueue
    buffer1 = (Angle.data[0] << 8) + Angle.data[1];
    buffer2 = (Angle.data[2] << 8) + Angle.data[3];
    // Convert the angle to degree
    DPen_Angle = (float)buffer1*9/50.0;
    DArm_Angle = (float)buffer2*9/50.0;
}
```

If the incoming message is the *Push_uC* one, another value which is equal to *RX_PUSB* (equal to 2) is saved on the *Wake_RxData*. This value (*RX_PUSB*) is the condition to run the code which recognizes the state of the IRP and the PWM value. Those states are also shown on the LCD.

```

// if receiving Push_uC Message
if (Check_Slot == RX_PUSB)
{
    if (Push_uC.data[2] == 1) // Button 3: Change input
    {
        switcher1 = FALSE;
    }
    if (Push_uC.data[3] == 1) // Button 6: finish changing input
    {
        switcher1 = TRUE;
    }
    if (Push_uC.data[0] == 1) // Button 1: Start application
    {
        switcher2 = TRUE;
    }
    if (Push_uC.data[1] == 1) // Button 2: stop application
    {
        switcher2 = FALSE;
    }
    buffer3 = (int) ((Push_uC.data[5] << 8 ) + Push_uC.data[6]);
    Motor_Power = (char) (buffer3/7.99);
    if (Motor_Power < 0) Motor_Power = -Motor_Power;
    if ((DPen_Angle > -163) && (DPen_Angle < 163)) Motor_Power = 25;
    if (!switcher2) Motor_Power = 0;
}

```

- **LCD_Display() Task**

This task displays all the information of the IRP which is processed by the *Data_Reception()* task. In order to display the long information, the program is constructed to show two LCD screen. The procedure to display information on LCD is depend on the information, it is converted to a single character or a string then one of the two functions *Dpc()* or *Dpstr()* is used to show them in the desired position on the LCD screen. When the previous character has finished, the next one is shown. The procedure continues to the last character. Two function *Dpc()* and *Dpstr()* belong to the *LCD-DOGS102* library. This code below is a part of this *LCD_Display()* task:

```

Clr();
Dpstr(0,0, "Welcome to IRP");
// Show the basic message
Dpstr(1,0,"State: "); // Show the word "State:"
// show the state
if ((!switcher2) && (!switch4)) Dpstr(1, 7, "Stop");
if ((!switcher2) && (switch4)) Dpstr(1, 7, "Power On");
if ((switcher2) && (!switch3)) Dpstr(1,7, "Swing Up");
if ((switcher2) && (switch3)) Dpstr(1,7, "Stable");
// Show the word "P_Ang:"
Dpstr(2,0,"P_Ang:");
// Show the word "A_Ang:"
Dpstr(3,0,"A_Ang:");
// show two angle values

```

The first code line is clear the LCD screen. The two character strings “Welcome to IRP” and “State:” are display at position (0,0) and (1,0) on the LCD screen. Then the if-condition is used to find what is the IRP’s

state. The state can be *Stop*, *Power UP*, *Swing UP* or *Stable*. The next two code lines shown the strings “*P_Ang:*” and “*A_Ang:*”. The *LCD_Display()* task code is shown in detail in the Appendix section.

4.5 PUSB-uC

This section will describe the code two task: *PWM_Calculation()* and *PushedButton_Detection()*. The former task receives an *Angles* message, calculate the PWM value and save on *Pusb_uC* message. The latter one detects whether any buttons are pushed or not, then the button signals are also saved on *Pusb_uC*. Then on the time slot 3, the *Pusb_uC* is transmitted. However, the pushed button event is a type of event trigger, which is different with the Time trigger structure in the whole system. To solve the problem, two struct variables *Pusb_uC0* and *Pusb_uC1* are created. The former variable only stores the reference input and PWM value and it is sent when there is no button pushed. The latter one stores the complete content including the control signal of start/stop button, change reference input and both the values of reference input and PWM. The second struct variable is only transmitted when there is a button pushed. This solution can avoid the conflict of the control signal of those control buttons.

- **PWM_Calculation() Task**

In order to calculate the PWM which is used to control DC-motor, there are needed many variables and parameters joining the process. The first group contains the controller parameters: *K1*, *K2*, *K3*, *K4* and *Ki*, which multiply with the IRP states and create the control signal. The second group consists of the states of the IRP, including *Arm_Up*, *Pen_Up*, *Arm_Vel*, *Pen_Vel* and *Control_Error*. The last groups include the remaining variables which are used as the buffer during the process. The list of variables and parameter are display in the figure below.

```
// system parameter
const float K1 = -3.92, K2 = 55.35, K3 = -5.86, K4 = 10.66, Ki = -1.41;
const float dt = 0.003;
// Local vars
float Arm_Up, Pen_Up, Arm_Vel, Pen_Vel, Last_ArmUp = 0, Last_PenUp = 0;
float Feedback, Moto_Stable, F_Ref_Input = 0;
float Current_Error, Control_Error = 0, SBData;
int buffer1, buffer2, Control_Value;
unsigned int count=0;
char C_Ref_Input, Oscillation = FALSE;
```

At time slot 2, when *Angles* message arrive the PUSB-uC the *TTCAN_Scheduler* task sent a semaphore to wake the *PWM_Calculation()* up. This task receives the data from the *Angles* message; then converts the data to the value of the two angles: *Pen_Angle* and *Arm_Angle*. The first two states of the IRP *Pen_Up* and

Arm_Up derives from offsetting Pen_Angle and Arm_Angle by 180 degree. The next step is to calculate two angular velocities Pen_Vel and Arm_Vel . The code is shown as following:

```
// receive data from the Angles message - the up angles
buffer1 = (Angle.data[4] << 8) + Angle.data[5];
buffer2 = (Angle.data[2] << 8) + Angle.data[3];
// Convert the angle to radian
Pen_Up = (float) (buffer1*3.142/1000);
Arm_Up = (float) (buffer2*3.142/1000);
// Velocities calculation
Pen_Vel = (Pen_Up - Last_PenUp)/dt;
Arm_Vel = (Arm_Up - Last_ArmUp)/dt;
// receiving new nominal Input?
```

If there is a new reference input which is inputted from Pushed Button Matrix, the new value is applied the variable F_Ref_Input .

```
// receiving new nominal Input?
if (xQueueReceive(NomInputQueue, &C_Ref_Input, 0))
{
    if (C_Ref_Input == 100) // signal to start oscillation
    {
        F_Ref_Input = 0.4;
        count = 0;
        Oscillation = TRUE;
    }
    else if (C_Ref_Input == -100) // signal to stop oscillation
    {
        F_Ref_Input = 0;
        Oscillation = FALSE;
    }
    else
    {
        // Convert Nominal value to radian
        F_Ref_Input = ((float) (C_Ref_Input))*0.01745; // signal to change reference input
        Oscillation = FALSE;
    }
}
```

Depend on the received value from the queue $NomInputQueue$, the F_Ref_Input will be assigned different value. If the received value C_Ref_Input is equal to 100, this is the signal to oscillate the IRP. If the value is “-100”, this signal stops the oscillation. If it is the other value, it means that is the new reference input of the arm; then it is converted to radian. If the IRP is set to oscillate, this followed code manipulates the magnitude and frequency of the oscillation. $Oscillation$ variable checks whether the oscillation command is available or not. The $count$ and F_Ref_Input variables set the period and magnitude of the oscillation respectively.

```
if ((Oscillation == TRUE)&&(count > 3300))
{
    if (F_Ref_Input > 0) F_Ref_Input = -0.4;
    else F_Ref_Input = 0.4;
    count = 0;
}
```

When the input of process is enough, the error signal and the feedback signal are calculated; then their total is the control signal in voltage. In order to calculate the PWM, the control signal divided by the supplied voltage (12V); then the result multiplies with the value of OCR3A (is equal to 799 – the maximum value of TCNT3), that is the PWM value.

```

Current_Error = Arm_Up - F_Ref_Input;          // Control Signal with nonzero ref. Input
Feedback = K1*Arm_Up + K2*Pen_Up + K3*Arm_Vel + K4*Pen_Vel;    //Feedback signal
// if the Pendulum angle is in the upright range
if ((buffer1 > -90) && (Pen_Up < 90))
    Control_Error = Control_Error + Ki*Current_Error;
else Control_Error = 0;
// total voltage apply to DC-motor
SBDData = -Control_Error*dt - Feedback;
// convert the voltage value to the value of OCR3B
Moto_Stable = (float) (SBDData/12.0);
Control_Value = (int) (Moto_Stable*799);

```

The PWM value then is saved by the data byte 5 and 6 of the message *Push_uC*. The Angle *Pen_Up* and the *Arm_Up* are saved for the next calculation.

```

// if there are no button pushed?
Push_uC0.data[5] = (unsigned char) (Control_Value >> 8);
Push_uC0.data[6] = (unsigned char) (Control_Value & 0x00ff);
// if any button is pushed?
Push_uC1.data[5] = (unsigned char) (Control_Value >> 8);
Push_uC1.data[6] = (unsigned char) (Control_Value & 0x00ff);
Last_PenUp = Pen_Up;
Last_ArmUp = Arm_Up;

```

• PushedButton_Detection() Task

In the *PUSB-uC*, there is a pushed button matrix with 12 buttons. All the buttons are controller by port D with four output pins and 4 input pins. In order to detect a pushed button, every output pins are set to low level “0” in turn, then the input pins are tested the level. If there is any pin shown the low level, it means at least a button is pushed. Depend on what output pin is set low and which input pin has low level, the pushed button is detected. In order to avoid the bouncing phenomenon when pushing and releasing the buttons, a *confidence level* is used to insure an actual button is pushed. The variables are applied in this task declared as following. The variables from *Press1* to *Press6* are used to check the buttons are really pushed and the variables from *Release1* to *Release6* are used to ensure the buttons are really released. Twelve variables are used to avoid the bouncing phenomenon with the insurance time is 8 milliseconds.

```

DDRF = 0b00001111;
DDRA = 0b11111111;
// Local vars
unsigned char i,k;
unsigned char Press1=0, Press2=0, Press3=0, Press4=0, Press5=0, Press6=0;
unsigned char Release1=0, Release2=0, Release3=0, Release4=0, Release5=0, Release6=0;

```

This task detects whether any button of six buttons is pushed or not. The principal to watch the pushed button is the same for all six buttons; therefore in this section a part of the task code will be presented.


```

PORTF = 0b11111110; // S1 .. S3
k = PINF; // read the PINF one time for the SYNC LATCH
if ((PINF & 0b00010000) == 0) // Button 1: Start button
{
    Press1++;
    if (Press1 > Confidence_Level) Press1 = Confidence_Level;
}
else if (Press1 == Confidence_Level)
{
    Release1++;
    if (Release1 > Confidence_Level) Release1 = Confidence_Level;
}
if ((Press1 == Confidence_Level) && (Release1 == Confidence_Level))
{
    for (i=0; i < 5; i++) Pusb_uC1.data[i] = 0;
    Press1 = 0;
    Release1 = 0;
    PORTA = 0b00000001; // turn on the respective LED
    Pusb_uC1.data[0] = 1; // Set the PushButton frame/message
    if (NominalInput < -90) NominalInput = -90;
    if (NominalInput > 90) NominalInput = 90;
    Pusb_uC1.data[4] = NominalInput; // save the reference input
    k = 1;
    xQueueSend(NomInputQueue, &NominalInput, 0); // send the ref. input to PWM_Calculation
    xQueueSend(Choose_Frame_Queue, &k, 0); // choose the changed frame to send
}

```

The first code line is set the output pin 0 to low level, then the input pin 4 is tested the level. If the level is low, it means the Start button is pushed. To ensure that is a real pushed (not bouncing phenomenon) the *Confidence_Level* is used to check the pushed period. During the pushed period the value of pin 4 is not changed, at that time it can conclude that the start button is pushed. The same principal is also applied for the release process. When the pushing and releasing process have actually happened, the actions for the pushed start button are performed. First, the data bytes on the *Pusb_uC1* message are reset then the respective LED is shined. The next step is to save the control signal on the first byte and nominal input of the fifth byte. The predefined values are sent to the *NomInputQueue* and *Choose_Frame_Queue* queues to activate the appropriate actions.

The entire codes of this task are shown on the Appendix part!

4.6 MOTO-uC

MOTO-uC has the responsibility to control the DC-motor swinging the pendulum up and keep at the upright position. In order to implement its duties, MOTO-uC receives two messages: *Angles* to determine where the pendulum is and *Pusb-uC* to get the control signal and PWM value. The DC-motor is controlled directly the PWM which has the frequency of 20 kHz. The PWM is performed by two Timer/Counter3 ISRs. The Two ISRs switch on or off the output level of Pin 4 and 5 of Port C.

Because the pendulum has two operation phases, Swinging Up and Upright Position, the control methods in two phases are different. In the swinging up phase, a constant energy is supplied the IRP every swinging period until the pendulum has enough energy to reach upright position. At that position, the control method is switched is the second one. In this method, the energy is transmitted to the IRP varied depend on the pendulum position; therefore PWM is adjusted to supplied the desired energy. This PWM value is calculated in the PUSB-uC by State Space method. As presented in the section 3.2.4, the Swinging up phases is controlled by *SwingUp_Controller()* task, and remaining phase is managed by *Upright_Controller()* task. The task *Data_Reception()* deals with the incoming messages.

4.6.1 Timer/Counter 3 Compare Match Interrupts

Timer/Counter 3 operates on Clear Timer on Compare Match (CTC) mode. In this operation mode, the Timer/Counter Register 3 TCNT3 increases from 0 to the value of Output Compare Register 3A OCR3A and is reset to “0”. The Output Compare Register 3B OCR3B is also enabled to change the output level of the Pins. The prescaler of Timer/Counter 3 is set to “1” therefore the OCR3A value is set to 799 to create the PWM frequency of 20 KHz. The ISRs and prescaler are set as the code following:

```
// Init Timer 3 for DCMotor PWM generation
TCCR3A = 0;           // CTC mode
TCCR3B = 0b00001001; // Prescaler: Clkio/1 (last 3 bits)
TCCR3C = 0;
TIMSK3 = 0b00000110; // Enable ISR on compA and compB
TCNT3 = 0;
OCR3B = 0;
OCR3A = 799;          // Time point to set PWM
```

The codes inside two ISRs are shown on the figure below. The Pin 4, 5 and 6 control the DC-motor. The pin 6 is set “1” to active the DC-motor. Those pin 4 and 5 are set 1 or 0 to change the voltage output of the pins. The Direction variable is used to control the direction of the DC-motor. If the Direction is equal to 1, the motor would run in clockwise direction and if it is equal to “2”, the motor will run in counter clockwise. If it is equal 3, the motor will standby.

```
ISR(TIMER3_COMPB_vect)
{
    PORTC &= 0b11001111;
}
ISR(TIMER3_COMPB_vect)
{
    xQueueReceiveFromISR(DirectionQueue, &Direction, 0);
    if (Direction == 3) PORTC &= 0b11001111;
    if (Direction == 1)
    {
        PORTC &= 0b11001111;
        PORTC |= 0b01010000;
    }
    if (Direction == 2)
    {
        PORTC &= 0b11001111;
        PORTC |= 0b01100000;
    }
}
```

4.6.2 DC-motor Control Tasks

This section introduces the three tasks: *Data_Reception()*, *SwingUp_Controller()* and *Upright_Controller()* in detail.

- **Data_Reception() Task**

Similarly to the LCD-uC, MOTO-uC receives two messages *Angles* and *Push_uC*. To reduce the process time, only the specific code executed when a predefined message arrives in a specific time slot by the if/else condition. Moreover, the values of the angles are in the integer form and run from -1999 to 1999 (the original value derive from the encoders). First, four variables are declared to perform in this task. There are two integer variables *buffer1*, *buffer2* which receive data from the *Angles* messages and two unsigned char variable – *Check_Slot* and *k* to check what kind of incoming message then give the appropriate response and control the direction of DC-motor.

```
int buffer1, buffer2;
unsigned char Check_Slot = 0;
unsigned char k = 3;
```

As a CAN message arrive, a predefined value is sent on the *Wake_RxData* queue to wake this task up. If the message is *Angles* message, the appropriate code calculates the three angles *Pen_Angle*, *Arm_Angle* and *Pen_Up*. *Pen_Up* is the angle when *Pen_Angle* is offset 180 degree. It is used to control the pendulum at the upright position. The respective code is shown as following.

```

// if receive angle message
if (Check_Slot == RX_ANGLE)
{
    // receive data from the Angle
    buffer1 = (Angle.data[0] << 8) + Angle.data[1];
    buffer2 = (Angle.data[2] << 8) + Angle.data[3];
    if ((buffer2 > 700) || (buffer2 < -700)) // safety condition
    {
        OCR3B = 0;
        xQueueSend(DirectionQueue, &k, 0);
        Start_Switch = FALSE;
    }
    // Get the angle of pendulum
    Pen_Angle = buffer1;
    Arm_Angle = buffer2;
    // Offset the Co-ordinate
    if (Pen_Angle < 0) Pen_Up = Pen_Angle + 1000;
    else Pen_Up = Pen_Angle - 1000;
}

```

When the second message – *Push_uC* arrive, the control signal variable – *Start_Switch* gets the data on the message. If the start button which is managed by *PUSB-uC* is pushed, that signal is saved on the first data byte of the message; that data is transferred to the *Start_Switch* variable then turn on the IRP operation. It is analogous for the stop button; then the *Start_Switch* shut the IRP down. After reading the data from *Push_uC* message, the *Pen_Up* is checked. If the angle belongs the upright position range, a semaphore is created to active the *Upright_Controller()* task. If not, another semaphore is sent to wake up the *SwingUp_Controller()* task. Then this task goes to blocked state to wait for next basic cycle. The task codes are shown on the figure below.

```

// if receive Push_uC message
if (Check_Slot == RX_PUSB)
{
    if (Push_uC.data[0] == 1) Start_Switch = TRUE;
    if (Push_uC.data[1] == 1)
    {
        OCR3B = 0;
        xQueueSend(DirectionQueue, &k, 0);
        Start_Switch = FALSE;
    }
    // trigger the Upright and SwingUp Controller
    if ((Pen_Up > -89) && (Pen_Up < 89)) xSemaphoreGive(Wake_Upright);
    else xSemaphoreGive(Wake_SwingUp);
}

```

- **SwingUp_Controller() Task**

This task's duty is to supply energy for the pendulum to swing up to the upright position. If the *Pen_Up* is out of the range (-90; 90), this task will be invoked. The PWM is set equal to 28% (OCR3B is equal 250) which is enough energy to swing up the pendulum (found by the experiment).

- *State machine*

In order to swing up, a state machine is applied which 5 states. The first state (case “0”) the IRP system is standby. After waiting for 2 seconds, it moves to state 2 (case “1”). In the second state, energy is supplied to rotate the arm in clockwise direction. As the Arm_Angle reach the value 32, the arm stops then the state moves to state 3 (case 2). In the third state, the DC-motor stops and wait for the pendulum moving back the lowest position. When the pendulum overcomes the lowest point, the state changes to state 4 (case 3). In the fourth state, the DC-motor is supplied energy again, and rotate the arm in counter-clockwise (opposite moving direction of the pendulum) back to the zero angle. As the Arm angle passes the zero position, the state goes to state 5 (case 4). In the state 5, the DC – motor is turn off again to wait the pendulum go back from the positive range. When the Pendulum angle decreases to zero, the IRP state jumps back the state 1, and do the same procedure. The Swinging up procedure repeats until the pendulum is in the upright range.

The rotation direction of the DC-motor is controlled by the *DirectionQueue* queue, a global variable. If the value 1 is sent to this queue, in the ISR the value is used to rotate the DC-motor in clockwise direction. If the value is 2, the DC-motor moves in the opposite direction. Other values on the queue will stop the DC-motor. The codes of the *SwingUp_Controller()* task are shown below.

```
void SwingUp_Controller(void * pvParameters)
{
    // Local vars
    unsigned char k = 3;
    unsigned char state= 0;
    for (;;) // Start of infinity loop
    {
        if(xSemaphoreTake(Wake_SwingUp, portMAX_DELAY))
        {
            if (Start_Switch) // if start button is pushed
            {
                OCR3B = 250; // set OCR3B to swing up
                switch (state)
                {
                    case 0: vTaskDelay(4000);
                            state = 1;
                            break;
                    case 1:
                            k = 1; xQueueSend(DirectionQueue, &k, 0);
                            if (Arm_Angle > 32) state = 2;
                            break;
```

```

        case 2:
            k = 3; xQueueSend(DirectionQueue, &k, 0);
            if (Pen_Angle > 0) state = 3;
            break;
        case 3:
            k = 2; xQueueSend(DirectionQueue, &k, 0);
            if (Arm_Angle < 0) state = 4;
            break;
        case 4:
            k = 3; xQueueSend(DirectionQueue, &k, 0);
            if (Pen_Angle < 0) state = 1;
            break;
        default:
            k = 3; xQueueSend(DirectionQueue, &k, 0); // Stop
            break;
    } // end of switch
} //end of Start_Switch condition
} // end of if (semaphore)
} // end of for() loop
}

```

- **Upright_Controller() Task**

This task is invoked when the Pen_Up angle is in the range (-90; 90). In order to control the pendulum in the upright range, the PWM is varied to supply enough energy to move the pendulum back the stable position. The PWM value is received from the data byte 5 and 6 of the *Push_uC* message. Depend on the value of PWM the DC motor rotates in the appropriation direction. The motor direction is also controlled by the value in the *DirectionQueue*. If the PWM value is too big, that could make the system vibration. To avoid that, a limited value is used to eliminate the harmful phenomenon. The codes of this task are shown on the figure below. In the codes, there is a section which is used to keep the IRP work in the desire range.

```

void Upright_Controller(void * pvParameters)
{
    char k = 3; // control the direction of DC-motor
    int SBData;
    for (;;) // Start of infinity loop
    {
        // check whether the data from Receive_Data is available or not?
        if(xSemaphoreTake(Wake_Upright, portMAX_DELAY))
        {
            if (Start_Switch) // if start button is pushed
            {
                SBData = (Push_uC.data[5] << 8) + Push_uC.data[6];
                if (SBData > 0)
                {
                    k = 1;
                    xQueueSend(DirectionQueue, &k, 0);
                }
                if (SBData < 0)
                {
                    k = 2;
                    xQueueSend(DirectionQueue, &k, 0);
                }
            }
        }
    }
}

```

```
    if (SBData == 0)
    {
        k = 3;
        xQueueSend(DirectionQueue, &k, 0);
    }
    if (SBData < 0) SBData = - SBData;
    if (SBData > 795) SBData = 795;
    if ((Arm_Angle > 668)|| (Arm_Angle < -668))
    {
        OCR3B = 0; //Safety condition
        k = 3;
        xQueueSend(DirectionQueue, &k, 0);
    }
    else OCR3B = (unsigned int) (SBData); // Change OCR3B, PWM
} // end of Start_Switch condition
} // end of if (xSemaphore ...)
} // end infinity loop
}
```

4.7 Parameters Estimation

In order to control the IRP system, the critical thing is to transmit precisely a needed energy to DC-motor to response the variation of the pendulum. The calculation process of the supplied energy depends completely on the system parameters, specifically the DC-motor parameters voltage and torque constants and the inertia momentum. Because of the lack of the information of those parameters, however, the estimation progress by the empirical method is taken place to find out what are the best appropriate system parameters. Moreover, the estimation of the weighting matrix which also affects the feedback signal is carried out. With good estimated parameters, the IRP can reach the upright position fast and stable.

4.7.1 The IRP parameters

In three parameters, voltage constant, torque constant and moment of inertia, the last one was approximate calculated in the student work [Bui, 2013]. However, there are some inertia moments of components in the IRP machine which is not able to calculate precisely such as the gear box, belt transmission. For the other parameters there is not any data which can be used to have a primitive calculation; Therefore the moment of inertia which derives from the student work is first used as a constant number to estimate the voltage and torque constant of the DC-motor. When the estimation process finds the good DC-motor constants which make the IRP operating better, then the found constant are used to estimate back the better moment of inertia value of the IRP. A data table sample which contains the information of empirical try is shown as followig:

	A	B	C	D	E	F	G	H	I	J	K
1	J	Kt = Km	K1	K2	K3	k4	time1	time2	time3		
2	0,002	0,0205	-0,791	34,636	-3,396	6,364	2,05	1,49	1,54		
3											
4	0,002	0,021	-0,791	29,318	-3,413	6,306	1,55	1,11	0,32		
5											
6	0,002	0,0215	-0,791	30,201	-3,48	6,421	0,32	0,39	1,52		
7											
8	0,002	0,022	-0,791	30,756	-3,55	6,541	0,25	1,12	1,01		
9											
10	0,002	0,0225	-0,791	31,706	-3,617	6,656	0,11	0,08	0,04		
11											
12	0,002	0,023	-0,791	32,738	-3,704	6,851	0,18	1,29	0,45		
13											
14	0,002	0,0235	-0,791	32,145	-3,767	6,905	no	0,09	0,08		
15											
16	0,002	0,024	-0,791	32,701	-3,838	7,026	0,46	0,41	0,51		
17											
18	0,002	0,0245	-0,791	33,259	-3,909	7,147	0,11	0,33	0,43		

Two DC-motor constants are equal in absolute value, therefore first both of them are tested with series of number with the step 0.005 and start from value 0.01. The Inertia moment J is fixed at value 0.002. The parameters were applied to the function *lqr()* on Matlab program, to calculate the control parameters K1, K2, K3 and K4. The control parameters then are used in the formula to calculate the feedback signal, control the DC-motor. Three column shows the times of three operations of the IRP machine. The longest operation time and oscillation of the pendulum are noticed to test more to ensure they are right value. When the best DC-motor constants are found, they are then fixed to estimate the inertia moment of the IRP as in the table following.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	J	Kt = Km	K1	K2	K3	k4	time1	time2	time3						
2	0,002	0,03	-1	37,84	-4,38	8,03	22,21	23,45	26,48						
3															
4	0,0021	0,03	-1	38,68	-4,37	8,03	24,52	23,45	24,56						
5															
6	0,0022	0,03	-1	39,44	-4,36	8,03	25,21	26,54	24,18						
7															
8	0,0023	0,03	-1	40,13	-4,35	8,03	40,12	42,18	48,56						
9															
10	0,0024	0,03	-1	41	-4,35	8,06	infinity	infinity	infinity	oscillation with strong amplitude					
11															
12	0,0025	0,03	-1	41,33	-4,34	8,03	infinity	infinity	infinity						
13															
14	0,0026	0,03	-1	43,05	-4,34	8,18	infinity	infinity	infinity						
15															
16	0,0027	0,03	-1	43,59	-4,34	8,19	18,21	23,21	26						
17															
18	0,0028	0,03	-1	44,09	-4,33	8,2	15,21	19,21	18,52						

As presented on the table above, with the value of inertia moment J is 0.025, voltage constant Km and torque constant Kt are 0.03, the IRP machine is able to work to infinity. However, their settling time is long and sometime oscillating. In order to reduce the response time, the LQR parameters are also to estimate to find the best parameters.

4.7.2 LQR parameters

In this IRP system, there is one input so the weighting matrix R reduces to a real number. The weighting matrix Q which depends on the states number of the IRP can be 4x4 or 5x5 dimensions. For the 4 states system the weighting matrix is called Q_1 and resemble for the 5 states system (the reference input is nonzero) the matrix is called Q_2 .

- **The estimation for the IRP without reference input**

The estimation procedure begins with the matrix Q_1 with assuming all the states of the IRP are the same penalty.

$$\underline{Q_1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R = 1$$

The matrix above is applied to calculate the feedback parameters then be varied if the IRP response is not good. The trend to change the coefficients of Q and R is: the smaller of the variation of a state is the bigger of the respective coefficient is. And the pendulum angle has the highest demand on variation, the arm angle has the medium requirement and the angular velocities have almost no penalty. After adjusting the matrix to have the best response, the weighting matrix Q has value:

$$\underline{Q_1} = \begin{bmatrix} 5 & 0 & 0 & 0 \\ 0 & 40 & 0 & 0 \\ 0 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 0.5 \end{bmatrix}$$

- **The estimation for the IRP with reference input**

Applying the same procedure, the matrix Q_2 , has value:

$$\underline{Q_2} = \begin{bmatrix} 5 & 0 & 0 & 0 & 0 \\ 0 & 40 & 0 & 0 & 0 \\ 0 & 0 & 0.2 & 0 & 0 \\ 0 & 0 & 0 & 0.2 & 0 \\ 0 & 0 & 0 & 0 & 2 \end{bmatrix}$$

With the structure design and the implementation as presenting above, the IRP machine was able to swing up and stay at the upright position stable. The result and response of the IRP are described in detail on the next chapter.

4.7.3 PWM and Sampling frequencies

The estimation of the appropriate PWM and sampling frequencies bases on some common rules. The first rule is the PWM frequency reverses to the output power of DC-motor, it means the higher the PWM frequency is, the less the DC-motor torque produces. The second one is for sampling frequency which is stated: the shorter the sampling period is, the better the behavior of the IRP is. However, the resolution of the encoders is 2000 values per revolution, the sampling period should not be too small.

The PWM frequency estimation started from 100 Hz on which the DC – motor was able to produce the maximal output torque. Then the frequency increased to have a better IRP response. At 100 Hz the IRP was not able to work stable and stayed at a specific position. The IRP operation show better from the frequency of 5000 Hz. From the frequency 10000 Hz the IRP was able to stay the upright position for a long time but it still vibrated strong. At the PWM frequency of 20000 Hz, the behavior of the IRP is the most stable and least vibration. With the higher PWM frequency, the operation of the IRP deteriorated and become unstable. The reason could be the output torque of DC –motor is not enough to control the pendulum. Therefore the PWM frequency of 20 kHz was chosen to work on the IRP machine.

The second sampling frequency estimation also began with 100 Hz which is equal to period of 10 milliseconds. At that period, the IRP was not able to stay at a predefine position but oscillation. Then the sampling period is shorten to the value of 4 milliseconds. The oscillation of the IRP is smaller corresponding to the decrease. At the period of 4 milliseconds, the IRP could work well without a reference input. If the IRP was desired to operate at arbitrary position, a more state is added to the system state vector (five elements). With new condition, the IRP operation is the best at period of 3 milliseconds. If the period is smaller than 2.5 milliseconds (equal to 400 Hz), the IRP system was not able to stay at the upright position.

5 Results and Discussion

After implementing the work and based on the IRP machine operation, this chapter presents the achievements and discusses several features of the IRP machine. In the first section, the responses of the IRP Simulink model and the machine are shown. In addition, the synchronization of the main tasks on four microcontrollers is also measured and described. In the second section several problems in the IRP machine which make its operation less stable are discussed, and what should be changed for a better operation IRP.

5.1 Results

The responses of the IRP nonlinear model and the IRP machine's operation are presented here. The IRP nonlinear model response is obtained by the Simulink model. The tasks operations in each microcontroller are measured by the MSO7014B Mixed Signal Oscilloscope. Meanwhile, the IRP machine operation is presented graphically by the collected data, which are obtained by USART port and Hterm device, then is plotted by software Gnuplot. Based on the operation measurement, the communication among the tasks and four microcontrollers is tested whether their operations complies with construction design or not. If the operations are synchronized and the IRP machine operates under the requirements, it can conclude that the structure design is good.

5.1.1 The nonlinear IRP model behavior

The responses of arm angle and pendulum angle of the IRP nonlinear model are shown below. The units on the vertical axis and horizontal axis of both graphs are radian and second respectively.

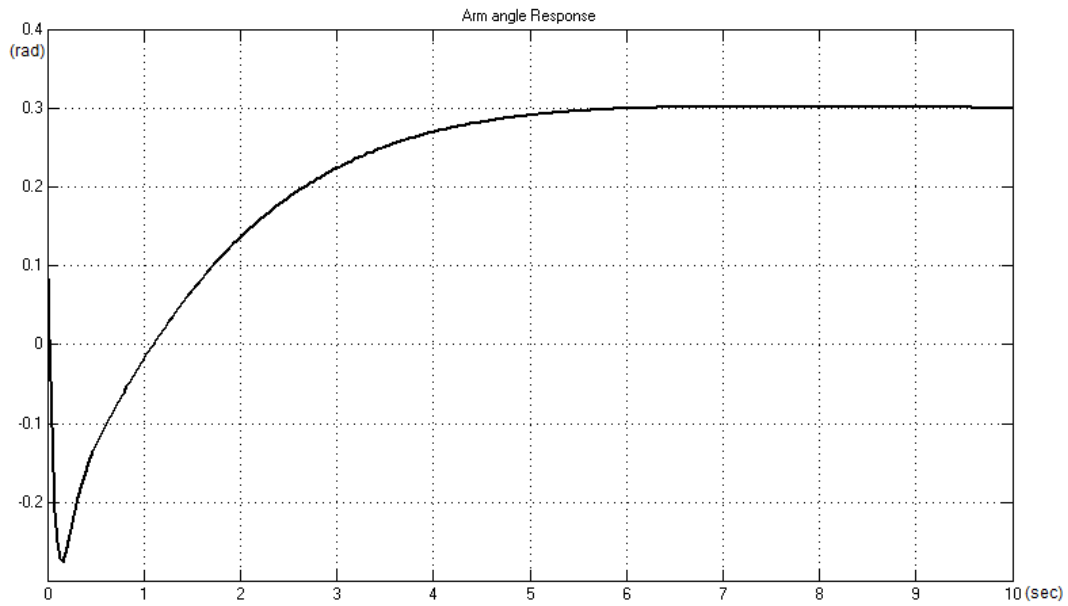


Figure 5.1: The Alpha/Arm angle

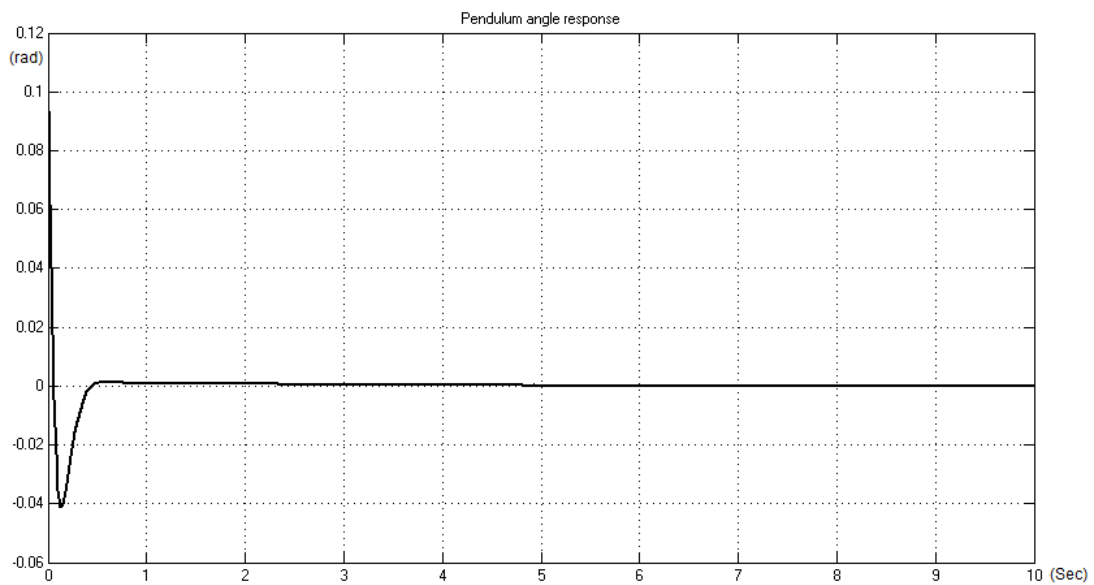


Figure 5.2: The Phi Angle/Pendulum angle

The two graphs display the response of the IRP nonlinear model at the upright position range. The initial position of both angles are 0.1 rad, the reference position of the arm angle at steady state is 0.3 rad. After near 5 seconds, the arm angle went to the desire position at 0.3 rad and stayed there. Meanwhile, the pendulum angle needs near 1 second to reach the upright position. Both the time response met the requirements which are stated in the chapter 1.

5.1.2 The IRP machine Operation

In this section, the synchronization of the communication among microcontrollers and task operations is tested. Then the pendulum position is presented and described.

- **Time scheduling TTCAN measurement**

In this section, the displaying of the microcontrollers' communication and tasks operation is performed by the device MSO7014B Mixed Signal Oscilloscope. It focuses on showing the critical information in one basic cycle (3 milliseconds). On each figure, the vertical axis shows the amplitude of the signal and the horizontal axis displays the time. The horizontal and vertical axes are divided into slots. A horizontal size is equal 0.5 milliseconds and the vertical one is equal to 5 V. In this configuration, the horizontal is able to display 10 slots (5 milliseconds) with the slot time name from slot zero to the fifth slot then going back zero. If there is nothing happened, the signal has value "0". If an action or a task is activated, the signal changes value to +5 V or -5V.

On the figure 5.3, it shows the operation scheduler of three actions: Sensor Data preparation, CAN message transmission and the CAN message reception. The yellow signal represents for the *Data_Preparation()* task. It is activated every 3 milliseconds on time slot "0". In the same time, the reference message is transmitted along the CAN bus. There are three CAN messages with green color transmitted in one basic cycle. The second message is *Angles* message and the last one is *Push_uC* message. In time slot "1", the ready *Sensor* message is sent via CAN bus and in time slot 2 three microcontrollers LCD-uC, PUSH-uC and MOTO-uC receive that message. The purple signal represents for the task *Data_Reception()* of MOTO-uC. All three actions above happened precisely as in the TTCAN scheduler.



Figure 5.3: the operation Plan of *Data_Preparation()* task, CAN message transmission and *Data_Reception()* task

The figure 5.4 presents the operation of three items. The yellow signal shows the transmission of CAN messages. After the first CAN message – *Angles* is sent in time slot “1”, it is received by the other microcontroller. In the PUSB-uC, the *Angles* message is received in time slot “2” then processed. The process is shown by the green signal. The result of process is the PWM value, which is sent in *Pusb_uC* message on the time slot “3”. The purple signal show the operation of the *Data_Reception()* task in the MOTO-uC. This task is waked up two times in a basic cycle by *Angles* and *Pusb_uC* message to control the DC-motor.

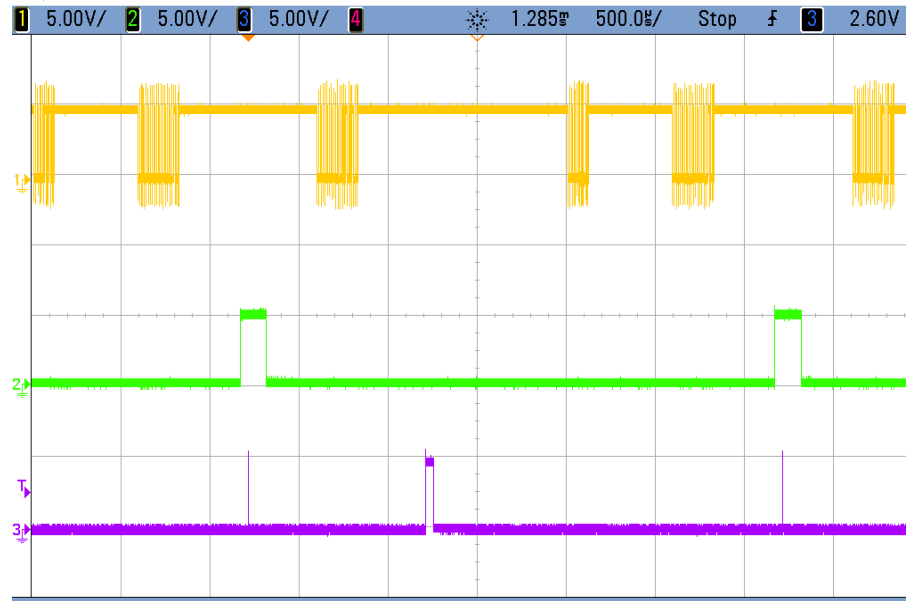


Figure 5.4: The Operation Plan of CAN message transmission, *PWM_Calculation()* task and *Data_Reception()* task

The figure 5.5 shows the time scheduler of three actions set: *PWM_Calculation()* task, *Data_Reception()* and *Upright_Controller()* in three color yellow, green and purple respectively. The first two actions are activated by *Angles* message. The *PWM_Calculation()* task then calculates the PWM value, which is used to control DC-motor to hold the pendulum at the upright position. When the calculation finishes, the data is saved on *Pusb_uC* message and transmitted on the next time slot. The *Data_Reception()* task on MOTO-uC is waked up again in two time slots later to receive the *Pusb_uC* message, then send a semaphore to wake up the *Upright_Controller()* or *SwingUp_Controller()* task. In figure 5.4, the task *Upright_Controller* is presented.

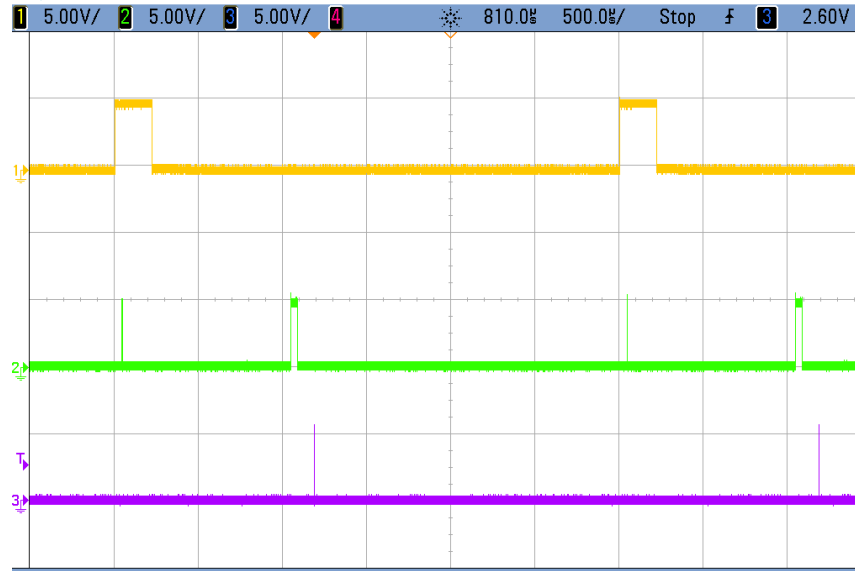


Figure 5.4: The Operation Plan of *PWM_Calculation()* task, *Data_Reception()* task and *Upright_Controller()* task

From the figures above, the microcontrollers operation are synchronized and strictly complied the TTCAN scheduler of the IRP system.

- **Position of the pendulum M and the arm**

The four figures (figure 5.6, 5.8, 5.9 and 5.10) which are plotted by *Gnuplot* software, presented the pendulum and arm position in two phases: swinging up and upright. The figure 5.6 shows the pendulum position in all working condition, swinging up then stay at upright position. The vertical axis presents the position of the pendulum and arm by the encoder values which are integer values from -1999 to +1999. And the horizontal axis shows the elapsing of time with the time unit in 0.5 milliseconds. The other figures show the arm positions with the different desired input. The figure 5.7 shows the swinging up phase and upright phase with the reference input at zero. The figure 5.8 presents the arm position when the reference input is 21 degree. The last figure shows the arm position oscillation when the reference input changes periodic.

In figure 5.6, starting from the zero position, the pendulum was supplied energy to swing up by *SwingUp_Controller()* task. After some period oscillation, it had enough energy to go to the Upright position range (the position is between (910; 1090) or (-1090; -910)), which was controlled by the *Upright_Controller()* task. When switching the roles (at time point 18500x0.5 ms), to avoid two control tasks the step up each other, a semaphore was used in the code. Depending on the initial condition, the pendulum reached the upright position which had position -1000 or 1000 then stayed at that position with small variation.

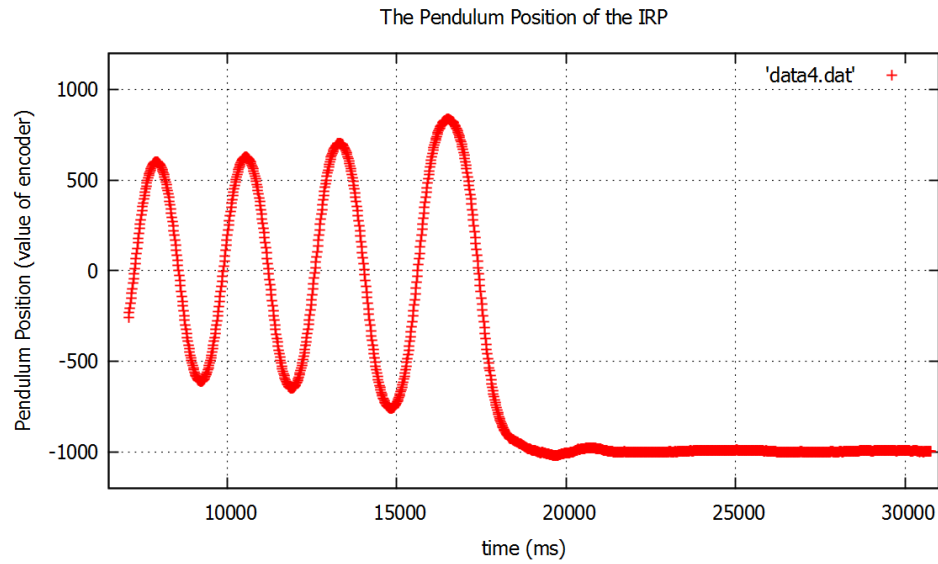


Figure 5.6: The Pendulum Position in the swing up phase and Upright phase

On the figure 5.7 showed the picture of the pendulum at upright position when the IRP machine was operating. Theoretically, the pendulum should work at that position forever.

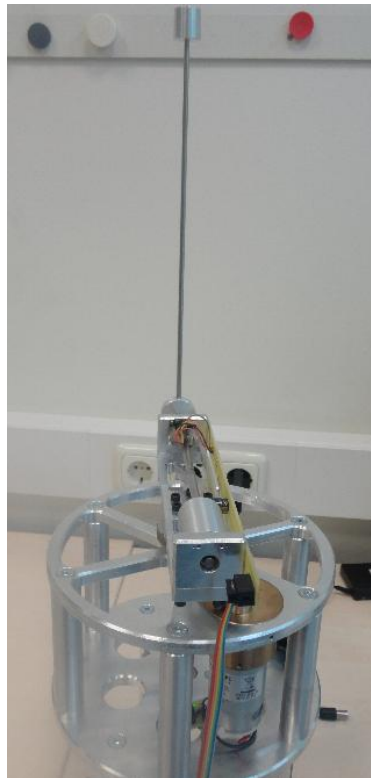


Figure 5.7: The IRP machine operation

On figure 5.8, the arm changed its position in period from approximate value 0 to 50 and vice versa by the DC-motor. By periodic movement, it supplied energy to the pendulum to swing up. After some time period, the pendulum went in the upright range (at time point 9000×0.5 ms), and the arm needed to response to hold

the pendulum at the top position. After a time, the IRP system reached the steady state and the oscillation of the arm became smaller and smaller. However, the arm was not able to stay at a precise position, which could be caused by the gearbox's backlash.

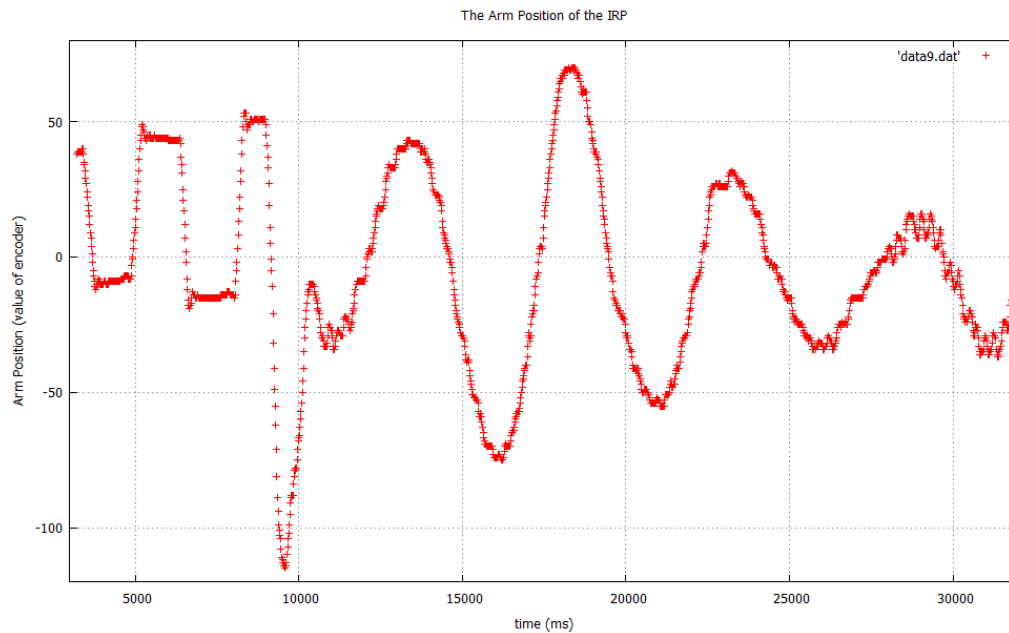


Figure 5.8: The Arm Position in the swing up phase and Upright phase

The Figure 5.9 presents the IRP's Arm position with nonzero reference arm input. The specific value in this figure is 21 degree, which is equal around the value 111 on the encoders. As the new reference input is applied, the arm moved slowly to the new position. The movement is not smooth but oscillating then reached the desired position at the time point -13000×0.5 ms. The time point is negative because the counter of FreeRTOS is defined as integer value. After reaching the nominal position, the arm stayed there with small oscillations.

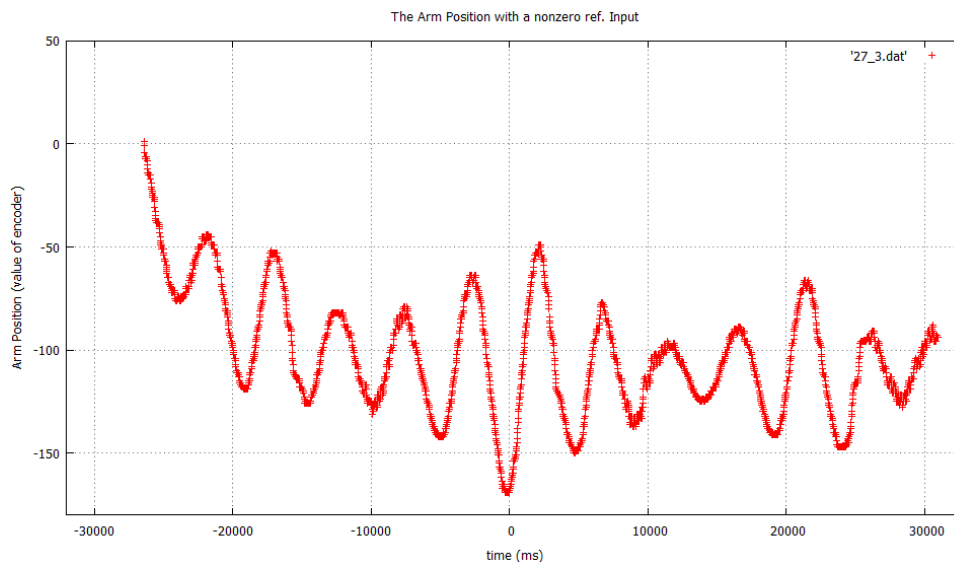


Figure 5.9: The Arm Position steady state with nonzero reference input

The Figure 5.10 presents the IRP's Arm position in oscillation. The magnitude is 22 degrees which is around the value 117 on the encoder. As the oscillation command is applied, the arm started oscillating. The figure shows a period of the arm oscillation, which is about 20 seconds.

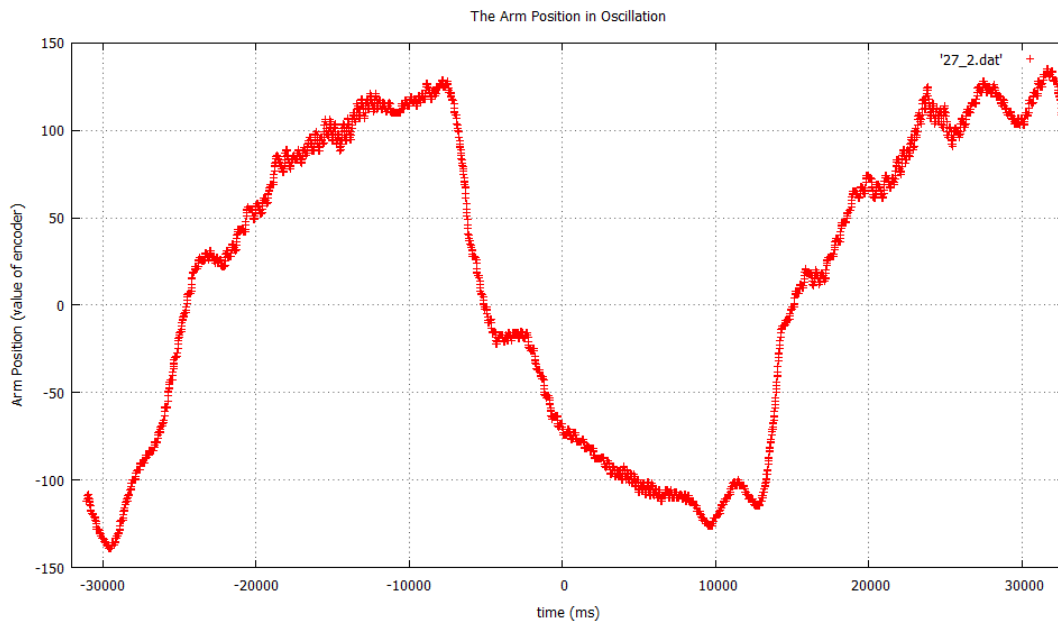


Figure 5.10: The Arm Position in oscillation

- **The operation of the circuit board (the chip)**

The figure 5.11 displays configuration of the circuit board (the chip) which control the whole IRP machine. It showed how the power supply, DC-motor cable and sensor signal cable were connected to the circuit board. When the circuit board is ready to operate, four green LEDs are lighted on to indicate that the four microcontrollers are available. The LCD shows the first screen, and the pushed button matrix is ready to receive the command.

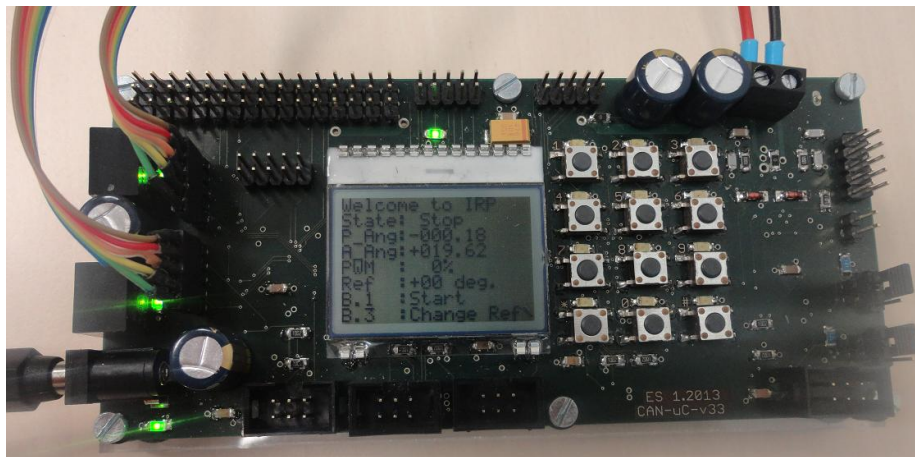


Figure 5.11: Configuration of the Chip

The figure 5.12 displays the two phases of the LCD, which showed the information of the IRP machine. The first screen (on the left) showed the greeting, state of the IRP, two output angles (Pendulum and Arm angle), the PWM value, reference input value and the buttons which control the IRP operation. On the second picture (on the right) which displayed the information and buttons which manage the IRP machine working at two programs, one for constant reference input, the other for varied reference input.

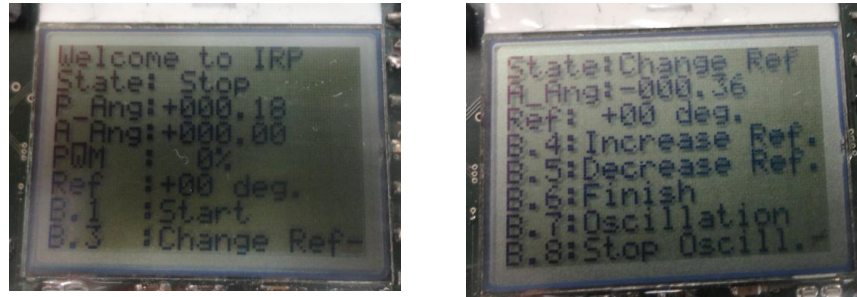


Figure 5.12: The two phases of the LCD in operation

5.2 Discussion

On the progress to control the IRP, there are many problems occurred. In order to reach the goals, the specific solutions are found to solve the obstacles. Besides that, there are several critical features of the IRP machine are established. This section will discuss about the critical issues and some suggestions which make the IRP's behavior better.

5.2.1 Sampling and PWM frequency

In the specific condition in this project, the IRP machine works well if the sampling and PWM frequency follow several configurations.

The sampling frequency should be in the range (250Hz; 400Hz). If it is smaller than 250Hz, the IRP operates with strong vibration. The strong magnitude of vibration can reduce the lifecycle of the transmission system (especially for belt transmission). If the sampling frequency is higher than 400Hz, the IRP operation is unstable. It is not able to hold the pendulum at the upright position.

The PWM frequency should belong to the range (10 kHz; 20 kHz). If this frequency is smaller than 10 kHz, the IRP system vibrates powerfully. In some case, it loses the stability. In the other hand, the PWM frequency is faster than 20 kHz cause the system instability. With so high frequency, the DC-motor on the IRP system can be not enough power to swing up and hold the pendulum.

5.2.2 Mechanical Structures

In this section, several remaining problems on the IRP machine are discussed. They are the classical problem in the mechanical field, which affects strongly to the IRP operation.

Firstly, the backlash in the gearbox is approximate 1.1 degree. As the pendulum is at the upright position, it varies with small angle. If the movement is less than 1.1 degree, the energy from DC-motor cannot be transmitted to the pendulum. The cumulative error increases time by time and can cause the ripple of the pendulum.

The second thing is the distance between the gravity center of the pendulum and the rotation center (the arm). If it is shorter, the behavior of the IRP machine would be better. The distance on the current IRP machine is 300 mm and it can make the IRP vibration and noise in operation.

The next issue is the parallel misalignment between the shaft of the gearbox and rotation axis of the arm. In the IRP machine, the error of two axes can cause the error energy transmission from gearbox to the arm. Moreover, the belt transmission can be damaged. To reduce the error, a calibration should be attached on the machine to manipulate the parallel misalignment.

And the last thing, if the DC-motor parameter such as voltage and torque constant can be calculated, the experiment time could be reduced dramatically. With all the estimation process, it costs several weeks to find the good values which are close to the true value.

All the discussed things above derive from by the empirical way or watching the behavior of the IRP with the noted data. It is true in the specific experimental condition of this IRP machine and can be not in general.

6 Conclusion

The master thesis is the second phase of the work which completes the IRP project from an idea to a functional machine. The work started with the modeling, simulation and control method. Then the following phase is to build the IRP machine and structures in microcontrollers which control the system operating in desired way. In spite of the drawbacks of the IRP machine, the minimal goals were done with accepted tolerance:

- It completed the IRP nonlinear Simulink model and an appropriate control method. The response of the IRP is positive.
- The constraint time to get the sensor signal and control the DC-motor which made the IRP working stable was defined.
- The combination of four microcontroller AT90CAN128s was enough speed for the IRP operation.
- The software program which controlled the IRP working at the upright position forever was accomplished.

Moreover, in some aspects the maximal goals were also reached.

- Two separate programs to control IRP in two phases Swing up and Upright were established.
- The Human – Machine Interface (HMI) was completed. The LCD displayed the IRP information, and the eight pushed buttons were used to input the IRP operation.
- The IRP was able to work at a nonzero position of the IRP's arm and could oscillate.
- The data of both the IRP's angles was transmitted to computer and shown graphically.

From this master thesis I – a Mechatronics student – had chance to improve my knowledge and understanding about the developing progress of a machine. I have a clearer overview about the role of each component in a machine, the critical features for a control system operation, how component's qualities affect the system behavior and solutions for them. From this project, I also know how a microcontroller operates and apply it to controls a motor or communicate with other microcontroller. I strongly believe that the things which I got from this master thesis project will assist me to develop my career.

I want to give my gratitude to Doctor Walter Lang due to his supports. With his advices in the microcontroller field, I could understand the issues or what could be the problem when several phenomena happened to the IRP machine. Based on that, I built the appropriate solutions.

Reference

- Atmel Corporation. (2008). *Datasheet AT90CAN32/64/128*. San Jose, USA: Atmel Corporation.
- Barry, R. (2009, October 15th). *freertos.org/RTOS.html*. (FreeRTOS, Producer, & Engineers Real Time Ltd) Retrieved January second, 2014, from FreeRTOS.
- Bui, D. H. (2013). *Modeling and Simulation of an Inverted Rotary Pendulum*. Siegen.
- CAN in Automation. (1992). *http://can-cia.com*. Retrieved 10 15, 2013, from *http://can-cia.com*
- Richard Barnett, Larry O'Cull, Sarah Cox. (2003). *Embedded C programming and the Atmel AVR*. Clifton Park, NY: Delmar.
- Thomas Führer, Bernd Müller, Werner Dieterle, Florian Hartwich, Robert Hugel, Michael Walther, Robert Bosch GmbH. (2000). *Time Triggered Communication on CAN*. Stuttgart: Robert Bosch GmbH.

Bibliography

- Florian Hartwich, Bernd Müller, Thomas Führer, Robert Hugel, Robert Bosch GmbH. (n.d.). *CAN Network with Time Triggerred Communication*. Reutlingen, Germany: Robert Bosch GmbH.
- Florian Hartwich, Bernd Müller, Thomas Führer, Robert Hugel, Robert Bosch GmbH. (n.d.). *Timing in the TTCAN Network*. Reutlingen, Germany: Robert Bosch GmbH.
- Kopetz, H. (2011). *Real time systems: Design Priciples for Distributed Embedded Applications*. New York: Springer.
- Obermaisser, R. (2010). *Event-triggerred and time-triggered control paradigms*. New York: Springer.
- Robert Bosch, G. (2013). *Time-Triggerred Controller Area Network User's Manual*. Gerlingen, Germany: Robert Bosch GmbH.
- Thomas Führer, Bernd Müller, Werner Dieterle, Florian Hartwich, Robert Hugel, Michael Walther, Robert Bosch GmbH. (2000). *Time Triggered Communication on CAN*. Stuttgart: Robert Bosch GmbH.

Appendix

- **TTCAN_Scheduler()** task

```
void TTCAN_Scheduler(void *pvParameters)
{
    struct CAN_message ref_message;
    static unsigned char Cycle_Number=0;
    unsigned char time_mark, base_mark, repeat, type, queue = 0;
    portTickType xLastWakeTime;
    xLastWakeTime = xTaskGetTickCount(); // integer type
    while(1)
    {
        if (Cycle_Time == 0 && TM == 1) //At time mark zero and is Time master
        {
            // Create RM to adjust the local time to global time
            ref_message.id=0x000; //RM's ID
            ref_message.length=2;
            ref_message.data[0]=(char) (TXtime >> 8); //Send the last transmitted time
            ref_message.data[1]=(char) TXtime;
            Can_TX(ref_message);
        }
        time_mark = pgm_read_byte (&Trigger[Cycle_Time][0]);
        base_mark = pgm_read_byte (&Trigger[Cycle_Time][1]);
        repeat = pgm_read_byte (&Trigger[Cycle_Time][2]);
        type = pgm_read_byte (&Trigger[Cycle_Time][3]);
        if (Cycle_Time == time_mark) //Reached the time mark?
        {
            //Check whether the frame provided in this cycle?
            if (((Cycle_Number-base_mark)%repeat==0 && Cycle_Number>base_mark)||Cycle_Number == base_mark)
            {
                switch (type)
                {
                    case 'R':
                    case 'r': queue = pgm_read_byte (&Trigger[Cycle_Time][5]); //receive message
                            if (queue != 0) xQueueSend(Wake_RxData, &queue, (portTickType) 0); break;
                    case 'S': //send message
                    case 's':
                    case 'A':
                    case 'a':
                    case 'T':
                    case 't':
                    default: break;
                }
            }
        }
        if (Cycle_Time == BASIC_CYCLE-1) //If the end of the basic cycle, cycle time reaches to zero
        {
            Cycle_Time = 0;
            if (Cycle_Number < MAX_CYCLE-1) Cycle_Number++;
            else Cycle_Number=0; //Another cycle or the end of the system matrix achieved?
        }
        else Cycle_Time++;
        vTaskDelayUntil(&xLastWakeTime, 1); // wait 0.5ms
    }
}
```

- **CAN ISR**

```
ISR(CANIT_vect)
{
    struct CAN_message buffer;
    unsigned char save_canpage, mob, i=1;
    static unsigned char sync = FALSE;
    static unsigned int last_ref, last_rx;
    static unsigned long int locdiff = 999*BASIC_CYCLE;
    save_canpage = CANPAGE; //insure CANPAGE
    mob = Get_Operating_Mob(CANSIT); //Get the Mob which its Interrupt is running?
    if(mob==0xFF) return;
    CANPAGE = (mob<<4); //Go to the Mob
    buffer.id = (unsigned int) (CANIDT1<<8); // Determine ID
    buffer.id |= (CANIDT2 & 0xE0);
    buffer.id >>= 5;
    if (CANSTMOB & 0b01000000) //Sending was successful?
    {
        if (buffer.id == 0x00) TXtime = CANTTC; // Get transmitted time of RM
        CANSTMOB &= 0b10111111; //Delete Transmit-Flag
        CANCDMOB &= 0b10111111; //disable Sending
    }
    if (CANSTMOB & 0b00100000) //Receive message successfully?
    {
        buffer.length = (CANCDMOB & 0x0F); //Determine amount of Databyte
        for (i=0;i<8;i++) buffer.data[i]=0; //clear buffer before receiving new data
        for (i=0; i<buffer.length; i++) buffer.data[i]=CANMSG; //Receive Data from the memory
        if (buffer.id == 0x00 && TM != 1) //receive RM and node is not Timemaster
        {
            TCNT1 = CANTIM - CANTTC; //reset local clock
            RXtime = CANTTC; //Save received time (SOF) of RM
            Cycle_Time = 1; //Reset time
            Ref_time = (buffer.data[0] << 8) + buffer.data[1]; //Read Reference-time
            if (sync) //Already received a RM?
            {
                locdiff = (RXtime - last_rx); //Determine local cycle duration
            }
            OCR1A = (unsigned int) (COMPARE*locdiff/(Ref_time - last_ref)); //Time Drift correction
        }
        last_ref = Ref_time; //save Ref_time and received time to calculate the next cycle duration
        last_rx = RXtime;
        sync = TRUE; //Flag for 'Reference-Message is received'
    }
    if (buffer.id == 0x0001) Angle = buffer; // Angles message?
    if (buffer.id == 0x0002) Push_uC = buffer; // Push_uC message?
    CANSTMOB &= 0b10101111; //Delete Receive-Flag
    CANCDMOB |= 0b10000000; //enable again Reception
}
if (CANSTMOB&0b00011111) Error(); //Failure? -> invoke Routine to troubleshooting
CANPAGE = save_canpage; // restore CANPAGE
}
```

- The Data_Reception() task on LCD-uC

```
void Data_Reception(void* pvParameters)
{
    // Local variables
    int buffer1, buffer2, buffer3;
    unsigned char Check_Slot = 0;
    for (;;)
    {
        // Receive new message?
        if (xQueueReceive(Wake_RxData, &Check_Slot, portMAX_DELAY))
        {
            if (Check_Slot == RX_ANGLE) // if receiving a angle message
            {
                // receive data from the AngleQueue
                buffer1 = (Angle.data[0] << 8) + Angle.data[1];
                buffer2 = (Angle.data[2] << 8) + Angle.data[3];
                // Convert the angle to degree
                DPen_Angle = (float)buffer1*9/50.0;
                DArm_Angle = (float)buffer2*9/50.0;
            }
            if (Check_Slot == RX_PUSB) // if receiving Pusb_uC Message
            {
                // Button 3: Change input
                if (Pusb_uC.data[2] == 1) switcher1 = FALSE;
                // Button 6: finish changing input
                if (Pusb_uC.data[3] == 1) switcher1 = TRUE;
                // Button 1: Start application
                if (Pusb_uC.data[0] == 1) switcher2 = TRUE;
                // Button 2: stop application
                if (Pusb_uC.data[1] == 1) switcher2 = FALSE;
                buffer3 = (int) ((Pusb_uC.data[5] << 8) + Pusb_uC.data[6]);
                Motor_Power = (char) (buffer3/7.99);
                if (Motor_Power < 0) Motor_Power = -Motor_Power;
                if ((DPen_Angle > -163) && (DPen_Angle < 163)) Motor_Power = 25;
                if (!switcher2) Motor_Power = 0;
            }
        } // end of xQueueReceive...
    } // end of for loop
} // end of task Convert_Angle
```

- * PWM_Calculation() task on PUSB-uC

```
//-----
// This function calculates the Controlled Data
//-----
void PWM_Calculation(void * pvParameters)
{
    // system parameter
    const float K1 = -3.92, K2 = 55.35, K3 = -5.86, K4 = 10.66, Ki = -1.41;
    const float dt = 0.003;
    // Local vars
    float Arm_Up, Pen_Up, Arm_Vel, Pen_Vel, Last_ArmUp = 0, Last_PenUp = 0;
    float Feedback, Moto_Stable, F_Ref_Input = 0;
    float Current_Error, Control_Error = 0, SBDdata;
    int buffer1, buffer2, Control_Value;
    unsigned int count=0;
    char C_Ref_Input, Oscillation = FALSE;

    for (;;) // Start of infinity loop
    {
        if (xSemaphoreTake(WakeTaskSe, portMAX_DELAY))
        {
            count++; // increase count value to oscillate the pendulum
            // receive data from the Angles message - the up angles
            buffer1 = (Angle.data[4] << 8) + Angle.data[5];
            buffer2 = (Angle.data[2] << 8) + Angle.data[3];
            // Convert the angle to radian
            Pen_Up = (float) (buffer1*3.142/1000);
            Arm_Up = (float) (buffer2*3.142/1000);
            // Velocities calculation
            Pen_Vel = (Pen_Up - Last_PenUp)/dt;
            Arm_Vel = (Arm_Up - Last_ArmUp)/dt;
```



```

// receiving new nominal Input?
if (xQueueReceive(NomInputQueue, &C_Ref_Input, 0))
{
    if (C_Ref_Input == 100)    // signal to start oscillation
    {
        F_Ref_Input = 0.4;
        count = 0;
        Oscillation = TRUE;
    }
    else if (C_Ref_Input == -100)    // signal to stop oscillation
    {
        F_Ref_Input = 0;
        Oscillation = FALSE;
    }
    else
    {
        // Convert Nominal value to radian
        F_Ref_Input = ((float) (C_Ref_Input))*0.01745; // signal to change reference input
        Oscillation = FALSE;
    }
}
if ((Oscillation == TRUE)&&(count > 3300))
{
    if (F_Ref_Input > 0) F_Ref_Input = -0.4;
    else F_Ref_Input = 0.4;
    count = 0;
}

Current_Error = Arm_Up - F_Ref_Input;    // Control Signal with nonzero ref. Input
Feedback = K1*Arm_Up + K2*Pen_Up + K3*Arm_Vel + K4*Pen_Vel;    //Feedback signal
// if the Pendulum angle is in the upright range
if ((buffer1 > -90) && (Pen_Up < 90))
    Control_Error = Control_Error + Ki*Current_Error;
else Control_Error = 0;
// total voltage apply to DC-motor
SBDData = -Control_Error*dt - Feedback;
// convert the voltage value to the value of OCR3B
Moto_Stable = (float) (SBDData/12.0);
Control_Value = (int) (Moto_Stable*799);
// if there are no button pushed?
Pusb_uC0.data[5] = (unsigned char) (Control_Value >> 8);
Pusb_uC0.data[6] = (unsigned char) (Control_Value & 0x00ff);
// if any button is pushed?
Pusb_uC1.data[5] = (unsigned char) (Control_Value >> 8);
Pusb_uC1.data[6] = (unsigned char) (Control_Value & 0x00ff);
// save data for the next calculation
Last_PenUp = Pen_Up;
Last_ArmUp = Arm_Up;
}
} // end infinity loop
} // end of task Calculate ControlledData

```

- **Data_Reception of MOTO-uC**

```
void Data_Reception(void * pvParameters)
{
    int buffer1, buffer2;
    unsigned char Check_Slot = 0;
    unsigned char k = 3;
    for (;;)
    {
        if(xQueueReceive(Wake_RxData, &Check_Slot, portMAX_DELAY)) //Receive angle message?
        {
            // if receive angle message
            if (Check_Slot == RX_ANGLE)
            {
                // receive data from the Angle
                buffer1 = (Angle.data[0] << 8) + Angle.data[1];
                buffer2 = (Angle.data[2] << 8) + Angle.data[3];
                if ((buffer2 > 700)|| (buffer2 < -700)) // safety condition
                {
                    OCR3B = 0;
                    xQueueSend(DirectionQueue, &k, 0);
                    Start_Switch = FALSE;
                }
                // Get the angle of pendulum
                Pen_Angle = buffer1;
                Arm_Angle = buffer2;
                // Offset the Co-ordinate
                if (Pen_Angle < 0) Pen_Up = Pen_Angle + 1000;
                else Pen_Up = Pen_Angle - 1000;
            }
            // if receive Push_uC message
            if (Check_Slot == RX_PUSB)
            {
                if (Push_uC.data[0] == 1) Start_Switch = TRUE;
                if (Push_uC.data[1] == 1)
                {
                    OCR3B = 0;
                    xQueueSend(DirectionQueue, &k, 0);
                    Start_Switch = FALSE;
                }
                // trigger the Upright and SwingUp Controller
                if ((Pen_Up > -89) && (Pen_Up < 89)) xSemaphoreGive(Wake_Upright);
                else xSemaphoreGive(Wake_SwingUp);
            }
        } //end of if (xQueue..)
    } // end of infinity loop
}
```

- **PushedButton_Detection() task**

```
//-----
// Detect_PushedButton detect pushed button then save the variation on PushButton
//-----
void PushedButton_Detection(void * pvParameters)
{
    DDRF = 0b00001111;
    DDRA = 0b11111111;
    // Local vars
    unsigned char i,k, Oscillation_Switch;
    unsigned char P1=0, P2=0, P3=0, P4=0, P5=0, P6=0, P7=0, P8=0;
    unsigned char R1=0, R2=0, R3=0, R4=0, R5=0, R6=0, R7=0, R8 = 0;

    for (i=0; i < 5; i++) Push_uC0.data[i] = 0;

    for (;;) // Start of infinity loop
    {
        PORTA = 0;
        PORTF = 0xff;
        PORTF = 0b11111110; // S1 .. S3
        k = PINF; // read the PINF one time for the SYNC LATCH

        // Button 1: Start button
        if ((PINF & 0b00010000) == 0)
        {
            P1++;
            if (P1 > Confidence_Level) P1 = Confidence_Level;
        }
        else if (P1 == Confidence_Level)
        {
            R1++;
            if (R1 > Confidence_Level) R1 = Confidence_Level;
        }
        if ((P1 == Confidence_Level) && (R1 == Confidence_Level))
        {
            for (i=0; i < 5; i++) Push_uC1.data[i] = 0;
            P1 = 0;
            R1 = 0;
            // turn on the respective LED
            PORTA = 0b00000001;

            // Set the PushButton frame/message
            Push_uC1.data[0] = 1; // send the start signal to the motor microcontroller

            if (NominalInput < -90) NominalInput = -90;
            if (NominalInput > 90) NominalInput = 90;
            Push_uC1.data[4] = NominalInput; // send the reference input to the LCD Microcontroller
            k = 1;
            xQueueSend(NomInputQueue, &NominalInput, 0); // send the reference input to PWM calculation
            xQueueSend(Choose_Frame_Queue, &k, 0); // choose the changed frame to send to other microcontroller
        }
    }
}
```

```

// Button 2: Stop button
if ((PINF & 0b00100000) == 0)
{
    P2++;
    if (P2 > Confidence_Level) P2 = Confidence_Level;
}
else if (P2 == Confidence_Level)
{
    R2++;
    if (R2 > Confidence_Level) R2 = Confidence_Level;
}
if ((P2 == Confidence_Level) && (R2 == Confidence_Level))
{
    for (i=0; i < 5; i++) Push_uC1.data[i] = 0; // reset the data on the frame
    // Turn on the respective LED
    PORTA = 0b00000010;
    // set the sending frame
    Push_uC1.data[1] = 1; // send a signal to turn off the DC motor
    Push_uC1.data[4] = NominalInput;
    k = 1;
    xQueueSend(Choose_Frame_Queue, &k, 0);
    P2 = 0;
    R2 = 0;
}

// Button 3: Change reference Input button
if ((PINF & 0b01000000) == 0) // if the third button is pressed?
{
    P3++;
    if (P3 > Confidence_Level) P3 = Confidence_Level;
}
else if (P3 == Confidence_Level) // if the third button is released?
{
    R3++;
    if (R3 > Confidence_Level) R3 = Confidence_Level;
}
if ((P3 == Confidence_Level) && (R3 == Confidence_Level))
{
    for (i=0; i < 5; i++) Push_uC1.data[i] = 0; // reset the data frame
    PORTA = 0b00000100;

    Push_uC1.data[2] = 1; // store the signal to show the change ref. Input screen
    Push_uC1.data[4] = NominalInput; // store the ref. input
    k=1;
    xQueueSend(Choose_Frame_Queue, &k, 0); // choose the Push_uC1 to send to other microcontroller
    P3 = 0;
    R3 = 0;
}

PORTF = 0xff;
PORTF = 0b11111101; // S4 .. S6
k = PINF; // read the PINF one time for the SYNC LATCH
// Button 4: Increase button
if ((PINF & 0b00010000) == 0)
{
    P4++;
    if (P4 > Confidence_Level) P4 = Confidence_Level;
}
else if (P4 == Confidence_Level) {
    R4++;
    if (R4 > Confidence_Level) R4 = Confidence_Level;
}
if ((P4 == Confidence_Level) && (R4 == Confidence_Level))
{
    for (i=0; i < 5; i++) Push_uC1.data[i] = 0;
    PORTA = 0b00001000; // Turn on the respective LED
    // set the PushButton Message
    NominalInput = NominalInput+1; // increase the reference input
    if (NominalInput > 90) NominalInput = 90; // check working condition
    Push_uC1.data[2] = 1; // store the signal to show the changed ref. Input screen
    Push_uC1.data[4] = NominalInput; // store the ref. input
    k=1;
    xQueueSend(Choose_Frame_Queue, &k, 0);
    P4 = 0; R4 = 0; // reset the check confidence_level
}
// Button 5: Decrease button
if ((PINF & 0b00100000) == 0)
{
    P5++;
    if (P5 > Confidence_Level) P5 = Confidence_Level;
}
else if (P5 == Confidence_Level) {
    R5++;
    if (R5 > Confidence_Level) R5 = Confidence_Level;
}
if ((P5 == Confidence_Level) && (R5 == Confidence_Level))
{
    for (i=0; i < 5; i++) Push_uC1.data[i] = 0;
    PORTA = 0b00010000; // Turn on the respective LED
    // set the PushButton Message
    NominalInput = NominalInput - 1; // reduce the ref. Input
    if (NominalInput < -90) NominalInput = -90; // check working condition
    Push_uC1.data[2]=1; // store the signal to show the changed ref. Input screen
    Push_uC1.data[4]= NominalInput; // save the ref. Input
    k=1;
    xQueueSend(Choose_Frame_Queue, &k, 0);
    P5 = 0; R5 = 0; // reset the check confidence_level
}
}

```

```

// Button 6: Finish button
if ((PINF & 0b01000000) == 0)
{
    P6++;
    if (P6 > Confidence_Level) P6 = Confidence_Level;
}
else if (P6 == Confidence_Level) {
    R6++;
    if (R6 > Confidence_Level) R6 = Confidence_Level;
}
if ((P6 == Confidence_Level) && (R6 == Confidence_Level))
{
    for (i=0; i < 5; i++) Push_uC1.data[i] = 0;
    PORTA = 0b00100000;
    Push_uC1.data[3] = 1; // store the signal to show the first screen
    Push_uC1.data[4] = NominalInput; // save the ref. Input
    xQueueSend(NomInputQueue, &NominalInput, 0); // send the new ref. Input for PWM calculation
    k=1;
    xQueueSend(Choose_Frame_Queue, &k, 0); // choose Push_uC1 frame to send to other microcontroller
    P6 = 0; R6 = 0;
}
PORTF = 0xff;
PORTF = 0b1111011; // check for button 7 and button 8
k = PINF; // read the PINF one time for the SYNC LATCH
// Button 7: Oscillation button
if ((PINF & 0b00010000) == 0) {
    P7++;
    if (P7 > Confidence_Level) P7 = Confidence_Level;
}
else if (P7 == Confidence_Level) {
    R7++;
    if (R7 > Confidence_Level) R7 = Confidence_Level;
}
if ((P7 == Confidence_Level) && (R7 == Confidence_Level))
{
    for (i=0; i < 5; i++) Push_uC1.data[i] = 0;
    PORTA = 0b01000000; // Turn on the respective LED
    Push_uC1.data[2] = 1; // store the signal to show the second screen
    NominalInput = 0;
    Push_uC1.data[4] = NominalInput; // store the ref. input
    Oscillation_Switch = 100;
    xQueueSend(NomInputQueue, &Oscillation_Switch, 0); // send the new ref. Input for PWM calculation
    k=1;
    xQueueSend(Choose_Frame_Queue, &k, 0);
    P7 = 0; R7 = 0;
}
// Button 8: Stop Oscillation button
if ((PINF & 0b00100000) == 0) {
    P8++;
    if (P8 > Confidence_Level) P8 = Confidence_Level;
}
else if (P8 == Confidence_Level) {
    R8++;
    if (R8 > Confidence_Level) R8 = Confidence_Level;
}
if ((P8 == Confidence_Level) && (R8 == Confidence_Level))
{
    for (i=0; i < 5; i++) Push_uC1.data[i] = 0;
    PORTA = 0b10000000; // Turn on the respective LED
    Push_uC1.data[2] = 1; // store the signal to show the second screen
    Push_uC1.data[4] = NominalInput; // store the ref. input
    Oscillation_Switch = -100;
    xQueueSend(NomInputQueue, &Oscillation_Switch, 0); // send the new ref. Input for PWM calculation
    k=1;
    xQueueSend(Choose_Frame_Queue, &k, 0);
    P8 = 0; R8 = 0;
}
Push_uC0.data[4] = NominalInput;
vTaskDelay(8);
} // end infinity loop
} // end of task Detect_PushedButton

```

- LCD_Display()

```

//-----
// This task show all the information on the LCD
//-----
void LCD_Display( void * pvParameters)
{
    int p=0, e=0;
    unsigned char sign, l=0;
    char t,h,z, k, f, n;
    float m=0, buffer1, buffer2;
    for (;;) // Start of infinity loop
    {

```

```

if (switcher1)
{
    Csr();
    Dpstr(0,0, "Welcome to IRP");
    // Show the basic message
    Dpstr(1,0,"State: "); // Show the word "State:"
    // show the state
    if (!switcher2 && (!switch4)) Dpstr(1, 7, "Stop");
    if (!switcher2 && (switch4)) Dpstr(1, 7, "Power On");
    if ((switcher2) && (!switch3)) Dpstr(1,7, "Swing Up");
    if ((switcher2) && (switch3)) Dpstr(1,7, "Stable");
    Dpstr(2,0,"P_Ang:"); // Show the word "P_Ang:"
    Dpstr(3,0,"A_Ang:"); // Show the word "A_Ang:"
    // show two angle values
    if ((DPen_Angle < -163)|| (DPen_Angle > 163)) switch3 = TRUE;
    else switch3 = FALSE;
    if (DPen_Angle == 0) switch4 = TRUE;
    else switch4 = FALSE;

    // Pendulum Angle on LCD
    if (DPen_Angle < 0) {
        sign = '-';
        buffer1 = -DPen_Angle;
    }
    else {
        sign = '+';
        buffer1 = DPen_Angle;
    }
    // Separate into two parts
    p = (int) buffer1;
    m = buffer1 - (float) p;
    // integer part
    t = (p / 100) + '0'; p = p - ((p / 100) * 100);
    h = (p / 10) + '0'; p = p - ((p / 10) * 10);
    z = (p % 10) + '0';
    // decimal/ fractional part
    m = m*100;
    e = (int) (m);
    f = (e / 10) + '0'; e = e - ((e / 10) * 10);

    n = (e % 10) + '0';
    // Show Integer part
    Dpc(2, 6, sign); Dpc(2, 7, t); Dpc(2, 8, h); Dpc(2, 9, z);
    // Show decimal part
    Dpc(2, 10, '.'); Dpc(2, 11, f); Dpc(2, 12, n);
    // Arm Angle on LCD
    if (DArm_Angle < 0) {
        sign = '-';
        buffer2 = -DArm_Angle;
    }
    else {
        sign = '+';
        buffer2 = DArm_Angle;
    }
    // Separate into two parts
    p = (int) buffer2;
    m = buffer2 - (float) p;
    // integer part
    t = (p / 100) + '0'; p = p - ((p / 100) * 100);
    h = (p / 10) + '0'; p = p - ((p / 10) * 10);
    z = (p % 10) + '0';
}

```

```

// decimal/ fractional part
m = m *100;
e = (int) (m);
f = (e / 10) + '0'; e = e-((e/10)*10);
n = (e % 10) + '0';
// Show Integer part
Dpc(3, 6, sign); Dpc(3, 7, t); Dpc(3, 8, h); Dpc(3, 9, z);
// Show decimal part
Dpc(3, 10, '.'); Dpc(3, 11, f); Dpc(3, 12, n);
Dpstr(4,0,"PWM : "); //Show the word "PWM:"
// Show Motor Power value on LCD
if (Motor_Power > 9)h = (Motor_Power/10) + '0';
else h = ' ';
Motor_Power = Motor_Power - ((Motor_Power/10)*10);
e = (Motor_Power % 10) + '0';
Dpc (4, 7, h); Dpc (4, 8, e); Dpc (4, 9, '%');
Dpstr(5,0,"Ref : "); // Show the word "Ref:"
// show the Nominal Input value
k = Push_uC.data[4];
if (k < 0) {
    sign = '-';
    k = -k;
}
else sign = '+';
z = (k / 10) + '0'; k = k - ((k/10) * 10);
e = (k % 10) + '0';
Dpc (5, 6, sign);

```

```

Dpc (5, 7, z);
Dpc (5, 8, e);
Dpstr(5, 10, "deg.");
// show the Start/Stop Button
if(switcher2) Dpstr(6, 0, "B.2 :Stop");
else Dpstr(6, 0, "B.1 :Start");
// show the Input Button
Dpstr(7,0, "B.3 :Change Ref"); // show the sentence B3 to change input
}
else // of if(switch1)..
{
    Csr();
    Dpstr(0,0, "State:Change Ref");
    // Show the word "A_Ang:"
    Dpstr(1,0,"A_Ang:");
    // Arm Angle on LCD
    if (DArm_Angle < 0) {
        sign = '-';
        buffer2 = -DArm_Angle;
    }
    else {
        sign = '+';
        buffer2 = DArm_Angle;
    }
}

```

```

// Separate into two parts
p = (int) buffer2;
m = buffer2 - (float) p;
// integer part
t = (p / 100) + '0'; p = p - ((p / 100) * 100);
h = (p / 10) + '0'; p = p - ((p / 10) * 10);
z = (p % 10) + '0';
// decimal/ fractional part
m = m *100;
e = (int) (m);
f = (e / 10) + '0'; e = e-((e/10)*10);
n = (e % 10) + '0';
// Show Integer part
Dpc(1, 6, sign); Dpc(1, 7, t); Dpc(1, 8, h); Dpc(1, 9, z);
// Show decimal part
Dpc(1, 10, '.'); Dpc(1, 11, f); Dpc(1, 12, n);
Dpstr(2,0,"Ref: "); // Show the word "Ref : "
Dpstr(3,0,"B.4:Increase Ref."); // Show the word "B.4:Increase Ref"
Dpstr(4,0,"B.5:Decrease Ref."); // Show the word "B.5: Decrease Ref : "
Dpstr(5,0,"B.6:Finish");
// shown the Ref. input
k = Push_uC.data[4];
if (k < 0) {

```

```
        sign = '-';
        k = -k;
    }
    else sign = '+';
    z = (k / 10) + '0';    k = k - ((k/10) * 10);
    e = (k % 10) + '0';
    Dpc (2, 5, sign);
    Dpc (2, 6, z);
    Dpc (2, 7, e);
    Dpstr(2, 9, "deg.");
    Dpstr(6,0,"B.7:Oscillation");
    Dpstr(7,0,"B.8:Stop Oscill.");
}
// check the LCD
l++;
if (l>3) l = 0;
switch (l % 4)
{
    case 0: Dpc (7, 17, '/' ); break;
    case 1: Dpc (7, 17, '-'); break;
    case 2: Dpc (7, 17, '\\'); break;
    case 3: Dpc (7, 17, '|'); break;
} // end switch
vTaskDelay (250);
} // end of infinity loop
} // end of task ShowOn_LCD
```