Real-time System Design

Robot Control with a Multi-task OS

17992969

Zhi Zhang

Objective

The objective of the assignment three is to master real-time system and be able to use it to implement an RTOS based robot system.

Methodology

To finish the whole project. The first and most important thing is to figure out the system requirements. Secondly, according to the system requirement, tasks and other resources are considered. A UML diagram would be very useful in this step. It be can used to show the structure of the whole system, and the relationship between each part of the system.

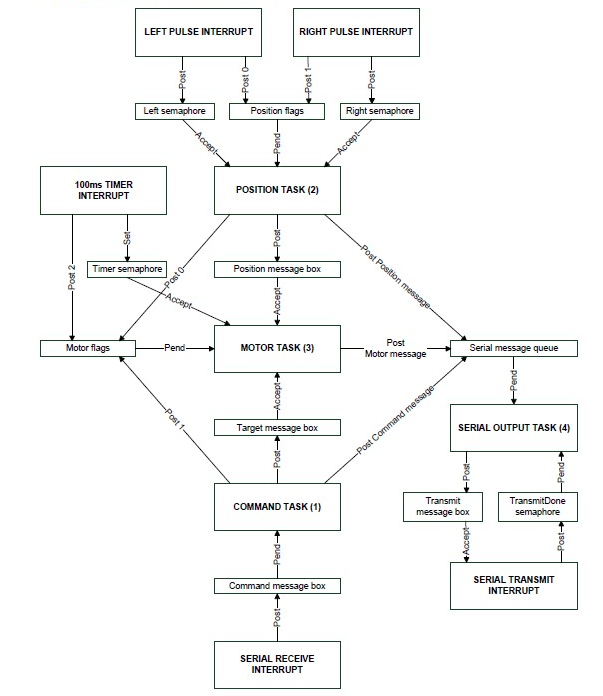
Once the system structure is defined, we can prepare to start to program. Sometimes, we don’t need to program from scratch. There may be some reference come. However, before starting from a reference code, doing some basic tests are necessary. These test helps us to acknowledge the reference code, both the advantages and disadvantages.

After all the things above, we can write the code. To write high-quality code. It is beneficial to think twice before typing the keyboard because it is usually having different choices. Another trick is to break the code into small parts. Once a small part of the code has finished, test it immediately. Otherwise, the problem can be much harder to find out in the future.

Design Description

The whole project can be divided into four different parts. They are command interpretation, position calculation, Motor control and serial communication. In a real-time system design, it is common to divide the whole system into different tasks. Each task responses for a specific work.

In this project, four core tasks are created except the first task who created the other tasks. These four tasks response to do command interpretation, position calculation, motor control and serial communication respectively. The structure of the whole system is shown in the picture below.



Apart from a timer interrupt function and a serial transmit interrupt function, there are three interrupt functions used to receive info from the environment. Two of them are used to sense the movement to the two wheels. Another one is used to receive the command from the serial receive port. The left and right pulse interrupt functions communicate with position task by two semaphores and a flag. The serial receives interrupt function communicate with command task by a message box. Motor task receivers communicate with position task and command task and a timer simultaneously.

Command interpretation

The command task is used to receive and interpret a command from the serial port. And then send a message to the motor task.

Command task does not read command from the serial port directly. A serial receivinbg interrupt function response to collect the data from the serial port and send it to a FIFO by using function OSQPost(). At the meantime, command receives data from the FIFO by calling function OSQPend(), and try to connect the data into a whole command. A command is received completely when a character “\r” is received. After that, it will be interpreted.

The interpretation part is almost the same as it was in assignment two. But to send the message to motor task needs more consideration. The following code demonstrates how a “GO” command is interpreted and send to the motor task as a message.

if (strncmp(command, "GO", 2) == 0)

{

n = sscanf(command +3,"%f,%f",&f1,&f2);

if (n == 2)

{

// get memory for message

pTarget = MemGet(MessageMem, &error);

// set value

pTarget->cmd = GO\_CMD;

pTarget->x = f1;

pTarget->y = f2;

// post valid commands to the Motor task

MboxPost(TargetMbox, pTarget);

OSFlagPost(MotorFlags, 2, OS\_FLAG\_SET, &error);

}

Function strcmp() is used to pick up a command which the system support. scannf() is used to extract the parameters embedded in the command line. These information will be filled in a memory block, which is allocated by function MemGet(). Once the message is ready, it will be sent out by mailbox function MboxPost. Since motor task receives various resources simultaneously, a flag is necessary. The flag for motor function is defined as “MotorFlags”.

Position Calculation

Position calculation is handled by position task, which calculates the current position according to the number of the pulses caused by the movement of the wheels. There are two active wheels in the robot, each wheel owns a pulse circuit to transform wheel movement into a pulse signal.

Two interrupt function are used to collect the pulse from two wheels separately. They share the same structure. In the function, it posts a semaphore whenever it triggered by a pulse. As the semaphore is a counter itself, it needs no counter variable in the interrupt function. Since the pulses can come from both the left and right wheels, “PositionFlags” are needed to supply more info for the motor task. The operation of notifying the position task in the interrupt function works as below.

// post semaphore for left wheel to Position task

OSSemPost(LeftPulseSem);

// set the event flag for the left wheel pulse

OSFlagPost(PositionFlags, 1, OS\_FLAG\_SET, &error);

The receiving part in the position task waits for a flag event at first, which gives the task the possibility of giving up CPU time. Then it read the semaphore with OSSemAccept function.

OSFlagPend(PositionFlags, 3, OS\_FLAG\_WAIT\_SET\_ANY + OS\_FLAG\_CONSUME, 0, &error);

leftCount = 0;

rightCount = 0;

while(OSSemAccept(LeftPulseSem))

leftCount++;

while(OSSemAccept(RightPulseSem))

rightCount++;

To make the task be able to give up the running right, using a flag is the only option. If OSSemPend is used for reading LeftPulseSem and RightPulseSem, the first OSSemPend would block the second OSSemPend. That would affect the accuracy of the second semaphore.

Once the position value is calculated, it should be sent to the motor task as soon as possible. The following code uses a mailbox to send the position message to the motor task. Similarly, a flag is set after the message is posted. In case the motor task cannot block itself properly.

pPos = MemGet(MessageMem, &error); // get memory for message

\*pPos = pos;

MboxPost(PositionMbox, pPos);

OSFlagPost(MotorFlags, 1, OS\_FLAG\_SET, &error); // post the flag

Motor Control

The motor control task receives the target position from command task and current position from position task, and then control the motors to move from the current position to the target position. It executes the command in a state machine. The following code shows how it updates the target position, current position and the state of the state machine.

OSFlagPend(MotorFlags, 7, OS\_FLAG\_WAIT\_SET\_ANY + OS\_FLAG\_CONSUME, 0, &error);

// check for a new position

pPos = OSMboxAccept(PositionMbox);

if (pPos != NULL)

{

pos = \*pPos;

OSMemPut(MessageMem, pPos);

}

// check for a new target command

pTarget = OSMboxAccept(TargetMbox);

if(pTarget != NULL)

{

target = \*pTarget;

OSMemPut(MessageMem, pTarget);

state = target.cmd;

state\_update = 1; //state to be updated

}

The current “state” is updated by “target.cmd ” whenever a new command is received. For some command like “GO”, it is necessary to supply an another variable to identify if entry of a state, there for state\_update is used.

In a multi-task system, tasks can never return unless it is deleted. So, the local variables can be used to track the system state. “state” and “state\_update” can be defined as none static variables.

The state machine is not very complicated. The following code shows how it works.

switch (state)

{

case STOP\_CMD:

OCR1A = 0;

OCR1B = 0;

state\_update = 0;

break;

case GO\_CMD:

if (Move(&pos,&target, state\_update))

{

state = STOP\_CMD;

}

state\_update = 0;

break;

}

Test Procedure

For a complicated system, there would be many cases to test. To make the test easier to expose problems in a system, the test sequence is very important.

In practical, tests can be from the bottom level to the top level and from outskirt to the core. Hardware related parts or interrupt functions are often the first choices. Then the communications between interrupt functions and tasks.

The debug port is usually the first hardware to be tested, because other test needs to show results through the debug port. In this project, the serial port can be used as a debug port. Therefore it is tested first. To test the serial, an echo code can be added into the command task. Once a command is sent out from the remote PC, it echoes immediately.

The second test is about the wheel pulses and the communication between pulse interrupt function and position task. To test these, a printing code can be added into the position task, which shows the value of the received pulses. Turn the wheel with a finger, the log should be printed out in the remote PC.

The third test can be focused on the communication of the motor task. To test this, the received current position and target position in the motor task can be print to the remote PC. Once a command is sent or a wheel is turned, the change of the positions should be observed from the print in the PC.

The fourth test is about motor control. Set different motor behaviour for different command in the state machine. Observe the motor behaviour while sending different commands.

After all the tests above, we can do the feature test. Try sending different commands randomly and observe how the robot react.

Conclusion

This project is implemented successfully. It achieved all the requirements assigned. In this project, I have had a better understanding of the real-time system design, especially on using “osflagxxx” “ossemaccept” and “osmboxaccept” functions.

When a task receives semaphore from only one task or interrupt function. It can use the ossempend to do it. However, when a task receives various semaphores from more than one tasks or interrupt functions. It is better to use the combination of “osflagpend” and “ossemaccept”. As the ossemaccept can reduce semaphore by one per time. Sometimes the “ossemaccept” should be called repeatedly to make the semaphore reduce to zero. Another way may be to extent the “ossemaccept” function.

In real-time system memory is scarce. To use the memory high efficiently, sometimes when sending a mail to the mailbox, we can check out an old mail, then update the content and post it back to the mailbox again. To slow the consumption of the FIFO memory, if it is possible, try to give a receiver higher priority than the transmitter when it is possible.