CHAPTER 6

RTOS IMPLEMENTATION FOR NONLINEAR SYSTEM

6.1 INTRODUCTION

i. This chapter describes the transformation for an existing non realtime process control system into realtime. The proposed system combines realtime scheduling with ZigBee wireless communication and monitoring. The methodology implements effective hardware realization in multitasking using RTOS concepts to suit critical applications and concurrent task handling in realtime environment.

ii. Inter task communication and data corruptions are major constraints in multi-tasking RTOS system. In this study, description about the solution for these issues with an example of realtime liquid level control system is given. Message queue and mail box are used to perform inter task communication to improve the task execution time and performance of the system. Critical section scheduling is used to eliminate the data corruption.

6.2 PROPOSED SYSTEM IMPLEMENTATION

The proposed system consists of a plant, processor with embedded µC/OS-II operating system, sensors and ZigBee based data transmission technologies. A realtime process control application approach is the host-server approach. A server uses general purpose OS/RTOS. Algorithm is running on host using µC/OS-II operating system. In this work for realtime implementation ARM processor is chosen, it has multi parameter acquisition and multi-level monitoring and supports networking.

Figure6.1 shows the proposed system architecture. It consists of two different sections,

- Client
- Server Sections

6.2.1 CLIENT SECTION

Client section has following modules(i) Host (ii) Controller (iii) Sampling device (iv) Control element. In client section the sensor node gathers the information such as water level,
motor status, in flow, out flow etc. The change in level is measured through a capacitive probe. Outputs of the plant are sampled at periodic intervals by the controller, a control algorithm applied to the samples, and the result of the control is produced at the output of the controller generally as a zero order hold signal. Plant parameters are updated to server at regular intervals through ZigBee IEEE 802.15.4.

![Figure 6.1 System Block Diagram](image)

6.2.2 SERVER SECTION

The server side consists of a status display and ZigBee module. It can display the parameters such as level, inflow, outflow, motor status etc. The set point is entered in server side which is transmitted host target through ZigBee. RTOS system is used in the server.

6.2.3 HOST-SERVER ZIGBEE COMMUNICATION

ZigBee RF modules interfaced to the host/server device through a logic-level asynchronous serial port. Through its serial port, the module can communicate with any logic and voltage compatible UART. Devices that have a UART interface can connect directly to the pins of the system. RTOS is used in server. Zigbee can operate either in a transparent data mode
(or) in a packet based Application Programming Interface mode (API) mode. In the transparent mode data coming into Data IN (DIN) pin is directly transmitted over air to the receiving mode without any modification. When RF data is received, the data is send out to (DO) pin. In the API mode the frame based API extends the level to which a host application can interact with the networking capabilities of the module all data entering & leaving the module is contained in frames that defines operation or events within module. ZigBee communication establishment between server & host is shown figure 6.2.

![Figure 6.2 Host Server communication](image)

1. RS232 Connector, 2a. PA2 of STM32 board, 2b. Vcc & ground, 3. ZigBee module
4. USB connector, 5. JTAG port

The connection details include,

- Connection between RS232 and STM32 board.
  b. Connect VCC pin and GND pin of RS232 to 5V and to GND.
- Connect RS232 TTL Converter to XBEE module using UART cable (male to male).
- Connect USB cable of the Zigbee module to the computer's USB port.
- Connect the JTAG port of the board to the computer's USB port using ULINK2/JLINK...
Embedded processor communication with ZigBee module through UART interface is shown in figure 6.3. Data enters the ZigBee module through UART interface DI pin as an asynchronous serial signal. The signal should idle high when no data is being transmitted. Each data byte consists of a start bit (low), 8 data bits (least significant bit first) and a stop bit (high).

The UART performs tasks, such as timing and parity checking, that are needed for data communications. Serial communications depend on the two UARTs to be configured with compatible settings (baud rate, parity, start bits, stop bits, data bits). Table 6.1 shows that the pin details of ZigBee - UART interface.

![Figure 6.3 Server-Host Zigbee Communication](image)

<table>
<thead>
<tr>
<th>Pin Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DI</td>
<td>Data in to ZigBee module</td>
</tr>
<tr>
<td>CTS</td>
<td>Clear To Send. Active low input to ZigBee module.</td>
</tr>
<tr>
<td>DO</td>
<td>Data out from ZigBee module</td>
</tr>
<tr>
<td>RTS</td>
<td>Request To Send. Active low output from ZigBee module</td>
</tr>
</tbody>
</table>

Table 6.1 ZigBee UART Interface Details

**6.3 REALTIME SYSTEM DEVELOPMENT CYCLE**

Embedded targets for nonlinear realtime applications cannot handle multiple inputs and multiple outputs, time constraints but perform sequentially. A real-time operating system (RTOS) is developed for real-time application. This implementation part describes algorithms for transforming a nonlinear control system to realtime. The development cycle starts with the design phase where in the design the following algorithms for ZigBee Communication between STM Boards. The steps used for design consideration includes.
ZigBee Module is initialized by sending AT commands to ZigBee through USART peripheral with baud rate 9600bps.

Use USART Flag USART_FLAG_TC to check whether the transmission has been completed and use USART Flag USART_FLAG_RXNE to check the reception buffer is not empty, wait for these flags to set and Transmit/Receive buffer through USART peripheral.

Then check the received response from XBEE by USART peripheral with "OK\r". Once XBee module is initialized, we can transmit/receive data from one module to another module.

### 6.3.1 HOST SOFTWARE DEVELOPMENT CYCLE

The μC/OS-II operating system is embedded host system to improve system reliability and stability. μC/OS-II operating systems are designed for embedded application. Most of the program is developed with C language. Because of the simple and well-knit structure, the μC/OS-II operating system becomes one of the most important embedded operating system. Table 6.2 shows the list of AT commands used to configure the ZigBee module.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘+++’</td>
<td>Enter into command mode.</td>
</tr>
<tr>
<td>ATID</td>
<td>set and read the PAN ID of the RF.</td>
</tr>
<tr>
<td>ATMY</td>
<td>set and read the 16-bit source address of the RF module</td>
</tr>
<tr>
<td>ATDL</td>
<td>set and read the lower 32 bits of the RF module's</td>
</tr>
<tr>
<td>ATBD</td>
<td>set and read the serial interface data rate</td>
</tr>
<tr>
<td>ATCN</td>
<td>exit the RF module from AT Command Mode</td>
</tr>
</tbody>
</table>

Table 6.2 ZigBee AT Commands
6.3.1.1 ZIGBEE INITIALIZATION

(1) First step is to define the following Global variables:

<table>
<thead>
<tr>
<th>MYID</th>
<th>DSTID</th>
<th>BAUDRATE</th>
<th>NETWORKID</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>11</td>
<td>9600</td>
<td>1111</td>
</tr>
</tbody>
</table>

Table 6.3 ZigBee Global Variable

(2) Second step is to initialize the ZigBee module using AT command set.

a) +++ ,

b) ATID = NETWORKID (network Id)

c) ATMY = MYID (Id of the this module)

d) ATDL = DSTID (Id of destination module)

e) ATBD = BAUD9600 (9600 baud rate)

f) ATCN (Exit Command Mode)

ZigBee Initialization Flowchart is shown in figure 6.4. Pseudo code for ZigBee Initialization is given below.
Figure. 6.4 ZigBee Initialization Flowchart
6.3.1.2 ZIGBEE DATA TRANSMISSION

For data transmission send start of file, data and end of file using ZigBee module through USART peripheral. Wait for USART flag USART_FLAG_TC (Transmission Complete flag) to set. The flowchart for transmitting data’s in ZigBee is given in figure 6.5.
Pseudo code for ZigBee Transmit data is given below.

```c
int SendFile(uint8_t *Buffer)
{
    int i, len;
    len = strlen((char *)Buffer);
    for(i=0; i<len; i++)
    {
        USART_SendData(USART2, Buffer[i]);
        while (USART_GetFlagStatus(USART2, USART_FLAG_TC) == RESET) {}}
    return 0;
}
```

Figure 6.5 ZigBee Data Transmission Flowchart

Pseudo code for ZigBee Transmit data is given below.
6.3.1.3 ZIGBEE RECEIVING DATA

For receiving data’s wait for USART flag USART_FLAG_RXNE (Receive data register not empty flag) to set, read the most recent received data by USART peripheral i.e., from USART Tx DR register using ZigBee module and check for start of file. If start of file is received, set the start flag else print failed to receive data. If start flag is set, load the data in USARTx DR register to reception buffer till the end of file. Wait for USART flag USART_FLAG_RXNE (Receive data register not empty flag) to set. Read the most recent received data by USART peripheral i.e., from USART Tx DR register using ZigBee module. Figure 6.6 shows the receive data flowchart. The pseudo code for receiving data’s is given below

```c
int ReceiveFile(uint8_t *Buffer)
{
    int i = 0, start_flag = 0;
    uint8_t ch;
    do{
        while (USART_GetFlagStatus (USART2, USART_FLAG_RXNE) == RESET);
        ch = USART_ReceiveData(USART2); //Receive data thru XBEE modem
        /* if start of file is sent set the start flag */
        if(ch == FILESTART)
        {
            start_flag = 1;
        }
        /* if data is received load it to buffer */
        else
        if(start_flag == 1)
        {
            Buffer[i++] = ch;
        }
    }while(ch != FILEEND); //checks till end of file
    if((ch == FILEEND) && (i> 0)) {
        Buffer[--i] = 0x00;
        /* write the received data in buffer to the file */
        printf("File Receive Success.. %ld bytes received\n",i);
        WriteToServer(Buffer);
        printf("****DISK CONTENTS****\n\n");
        display(path);
        printf("\nFile has been received successfully\n");
        return(0);
    }
    printf("Failed to receive File\n");
    return 0;
}
```

Figure 6.6 ZigBee Receive Data Flowchart
6.3.1.4 SENDING COMMAND

Send AT commands to ZigBee module through USART peripheral which is connected to this module with baud rate 9600. Wait for USART flag USART_FLAG_TC (Transmission Complete flag) to set. The flowchart for ZigBee commands for sending is given in figure 6.7.

Figure 6.7 ZigBee Send Command Flowchart

6.3.1.5 CHECK RESPONSE

Wait for USART flag USART_FLAG_RXNE (Receive data register not empty flag) to set. Receive the response from ZigBee module by USART peripheral. Check the received response with "OK\r". The flowchart for ZigBee commands for checking response is given in figure 6.8. Pseudo code for ZigBee check response is given below.
int CheckResponse (void)
{
    uint8_t resp[5], ch;
    int i = 0;
    do
    {
        while (USART_GetFlagStatus (USART2, USART_FLAG_RXNE) == RESET); // Loop until RXNE = 1
        ch = USART_ReceiveData (USART2); // Receive the response from XBEE modem (from USART2)
        resp[i++] = ch;
    } while (ch != ''); // Response is “OK\r” for all the commands
    resp[i] = 0;
    if (strcmp (resp, "OK\r") == 0) // Check the response for each command before continuing with the next command.
        return (0);
    return (1);
}
6.3.2 PORTING OF µC/OS-II FOR NONLINEAR SYSTEM

At startup, the system runs initialization functions by itself by using restore system state and set initial value of each parameter. Initialization task includes

1) Initializing all data structure,
2) Allocating memory space for stack
3) Establishing semaphore message queue of inter-task communication,
4) Create tasks and set different priority.

The resources are allocated for the tasks defined in the application, the scheduler is started then it schedules the task in preemptive manner. The system turns fully operational when the initialization is done by using specified OS function. Figure 6.4 shows the startup flow of µC/OS-II.

![Flowchart of µC/OS-II Initialization](image)

**Figure 6.9 µC/OS-II Initialization**

Initialization all data structures has following steps,

1) Disable interrupts and user initialization
2) Using kernel initialization function ‘OSInit()’ initialize interrupting nesting counter, Task counter, Context switch counter, statistics, TCB, event lists and ect.
3) Create application task
4) Start the execution of the OS using kernel function ‘OSStart()’.

6.3.2.1 HOST SYSTEM TASK

In the Host system following tasks are introduced

1) Read Status (Task1),
2) Display Process Status (Task2),
3) Transmit data status (Task3)
4) Write set point (Task4)

All tasks are running in preemptive round robin scheduling algorithm. Task1 to Task3 are running round robin. Tasks such as writing set point and transmit status data to server are running on the interrupt received from the ZigBee. When task1 is running all other tasks are in waiting state. The tasks become active when the scheduler schedules it. Write set point message gives the values liquid level and outflow. When host system receives set point through ZigBee, it initiates write liquid level and outflow values to the controller. When host receives status read message from server, it response to server through ZigBee with values of liquid level, outflow and motor ON/OFF status etc. Pseudo code for µC/OS-II startup and task creation is given below.

```c
int main (void)
{
    EUARTinit();// Init UART controller
    EUARTputString("nEntering main() now!
");
    OSInit();     // InituCOS kernel
    OSTaskCreate(task1, (void *)0, &(TaskStk[0][TASK_STK_SIZE - 1]), 1);
    OSTaskCreate(task2, (void *)0, &(TaskStk[1][TASK_STK_SIZE - 1]), 2);
    OSTaskCreate(task3, (void *)0, &(TaskStk[2][TASK_STK_SIZE - 1]), 2);
    OSTaskCreate(task4, (void *)0, &(TaskStk[3][TASK_STK_SIZE - 1]), 2);
    OSStart();    // Start multi-tasking now
    return 0;    // Never comes here
```
µC/OS-II provides various functions to initialize and activate new tasks. A task can obtain information about itself or other tasks. This information can be used to know what the task is doing at a particular time. A task at an instance can be in one of the states, waiting, ready to run and running that are controlled by the scheduler. The task states are such as Dormant, Ready, Running, Waiting and ISR. Task Information is useful for debugging and monitoring parameters. Tasks become "ready" after they are created. Each task is assigned a priority. The lower the priority number, the higher the priority of the task. Always the highest priority task that is ready to run should be executed for realtime. The API functions used are listed in table 6.3.

<table>
<thead>
<tr>
<th>µC/OS II API</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_TASK_Name_EN</td>
<td>Enabling Task Name</td>
</tr>
<tr>
<td>App_Task1Create()</td>
<td>Create the Application Task</td>
</tr>
<tr>
<td>OSTaskCreate()</td>
<td>Creating a Task</td>
</tr>
<tr>
<td>OSSchedLock()</td>
<td>Prevents tasks rescheduling</td>
</tr>
<tr>
<td>UART0_Send string()</td>
<td>Send the string through UART 0</td>
</tr>
<tr>
<td>Static OS_STK align</td>
<td>Aligning the state using the stack in tasks</td>
</tr>
<tr>
<td>OSTaskChangePrio()</td>
<td>Changes the priority of task</td>
</tr>
<tr>
<td>OSSchedUnlock()</td>
<td>Re enables tasks scheduling</td>
</tr>
<tr>
<td>UART0_Init()</td>
<td>Initialize the UART Port</td>
</tr>
<tr>
<td>OS_FLAG_CONSUME</td>
<td>Consume the Arguments</td>
</tr>
<tr>
<td>OS_Task_delete()</td>
<td>Delete a Task</td>
</tr>
<tr>
<td>OS_Task_Suspend()</td>
<td>Suspend a particular Task</td>
</tr>
<tr>
<td>OS_Task_Resume()</td>
<td>Resume the Task</td>
</tr>
<tr>
<td>OS_FLAG_PEND</td>
<td>Arguments are Waiting</td>
</tr>
<tr>
<td>OS_FLAG_Wait_SETALL</td>
<td>Set all Values While pending</td>
</tr>
<tr>
<td>OS_ENTER_CRITICAL</td>
<td>Enter to Critical</td>
</tr>
<tr>
<td>OS_EXIT_CRITICAL</td>
<td>Exit From Critical</td>
</tr>
</tbody>
</table>

Table 6.4 Common RTOS functions

6.4 RTOS IMPLEMENTATION OF MULTITASKING USING µC/OS-II

Realtime liquid level system consists of a conical tank, Host controller and ZigBee module. Conical tank has a liquid level sensor, inflow and outflow rotary sensor. These sensor outputs are connected to controller in host. Host has controller and a ZigBee module. ZigBee module is used to communicate with server. Controller calculates final action values and drive to valve control inputs in conical tank. The hardware implementation of the realtime control system is shown in Figure 6.10 to 6.12.
Figure 6.10 Host system and conical tank setup 1. Conical Tank 2. Host controller

Figure 6.11 Host Controller module 1. Host ZigBee 2. Controller
6.4.1 DEVELOPMENT CYCLE

A non real-time liquid level control system is transformed to real time using multiple tasks. In order to manage the various tasks, Priority based Preemptive Task Scheduling algorithm in RTOS is used. Each task in an application is assigned a priority, with higher priority values representing the need for quick response. In modern RTOS multitasking is a technique used for enabling multiple tasks to share a single processor. It is simply the ability to run two or more independent tasks on one CPU (appears to be at the same time) and running concurrently.

The following tasks are identified:

- Tracking input to Controller (Task1)
- ZigBee Communication (Task2)
- Error handling (Task3)
- Monitor process value (Task4)

**TRACKING INPUT TO CONTROLLER**

The function of Task1 is getting the setpoint values from server and writes these data into controller. Flowchart for setpoint tracking is given in figure 6.13
• **ZIGBEE COMMUNICATION**

Task2 access the process status values namely inflow, level and outflow. The collected process status values are send to the server through Zigbee communication. Task2 flowchart is given in figure 6.14.

• **ERROR HANDLING**

The function of Task3 is processing the error and deciding the control output for action. When this task is started it gets the process value such as level from server then it wait for an interrupt from server. Once it receives an interrupt from the server, reads the process value from plant for final action. Task3 flowchart is given in figure. 6.15.

Figure 6.13 Flowchart for setpoint tracking
Figure 6.14 Flowchart for Status Communication

- Get Process value level from server
- Wait for interrupt from server
- Access inflow, level, outflow values
- Transmit Status values to server
- End

Figure 6.15 Flowchart for Error Handling

- uctsk_error
- Get Process value level from server
- Wait for interrupt from server
- Read process values
- Process the level
- End
• **MONITOR PROCESS VALUE**

  Task 4 is used to display the data’s such as set value, outflow, level, motor on/off condition etc. in the tank monitoring screen. When this task is started it will wait for error signal interrupt. Task4 flowchart is shown in Figure 6.16.

![Flowchart for Status Display](image)

**Figure 6.16 Flowchart for Status Display**

6.4.2 **ALGORITHM FOR CRITICAL ACTIVITY**

- For critical section implementation of µC/OS II has interrupt disabling and enabling functions that execute at entering and exiting the section respectively. (OS_ENTER_CRITICAL and OS_EXIT_CRITICAL).
- To achieve significant enhancement in the task execution time and prevention of other task to run in-between, these application programming interfaces are incorporated in RTOS under critical environment that provides the interface between the application software and system software.

  Disabling APIs result in non RTOS implementation and increase the task execution time. Once it receives the interrupt it will display process status data. Task4 job is considered critical as it should not be preempted, when it is reading and displaying process value. A critical section of code, also called a critical region, is code that needs to be treated indivisibly. Once the section of code starts executing, it must not be interrupted. To ensure this, interrupts are typically disabled before the critical code is executed and enabled when the critical code is finished. Disabling the interrupts before a critical section starts executing and enabling interrupts is a powerful option for solving shared resource problems. For critical section implementation RTOS
has interrupt disabling and enabling functions that execute at entering and exiting section respectively (OS_ENTER_CRITICAL and OS_EXIT_CRITICAL). To achieve significant enhancement in the task execution and prevention of other task to run in between, these application programming interfaces are incorporated in RTOS under critical environment that provides the interface between the application software and system software. Task4 with critical implementation flowchart is given in Figure 6.17.

**Figure 6.17 Flowchart for monitoring process value with Critical Section**

If the system needs to respond to the host computer, it will run the ZigBee interrupt program. µC/OS-II operating system is a preemptive kernel that means the task that has a higher priority can ready to run. Therefore, we need to set the task that needs quick response in ready state after entering the interrupt program, so that higher priority task can be run right after exiting the interrupt program. However, it's difficult for the traditional system without µC/OS-II to control when a task can run. Accordingly, this system based on µC/OS-II has good real-time capability. The pseudo code for entering and exiting critical section is given below,
6.4.3 INTERPROCESS COMMUNICATION

Through kernel services message can be sent to a task. A message mailbox also called a message exchange. It is typically a pointer size variable through a service provided by the kernel. A task (or) an ISR can deposit a message into the mail box. Through a service provided by the kernel, one or more tasks can receive messages. Both the sending task and receiving task will agree as to what the pointer is actually pointing to. Each mail box is associated with a waiting list, in case more than one task desires to receive message through the mail box. Communication from Task3 to Task2 and Task4 using mailbox is shown in figure 6.18. Task2 and Task4 has highest and equal priority task and Task1 has low priority. Task3 has medium level priority. A mailbox is created using OSMboxCreate() function to communicate process status values to Task2 and Task4. In Task3 read inflow, level and outflow values from sensors in plant and post a message to mailbox using OSMboxPost() function. Task2 and Task4 are in waiting state for

```c
static void uctsk_display (void *pdata)
{
    #if OS_CRITICAL_METHOD == 3 /* Allocate storage for CPU status register */
    OS_CPU_SR cpu_sr;  //This is necessary when using Critical section
    #endif
    unsigned int i, j, size, k;
    uint8_t err = 0;
    UART0_SendString("\r\nTask 2(uctsk_display) is Running\r\n");
    bmp = OSMboxPend(MBox, 100, &err); // Waiting for mail from mailbox
    OS_ENTER_CRITICAL(); // Disable interrupts for Critical section
    UART0_SendString("\r\nDISPLAY DATA IS :");
    size = sizeof(bmp);
    for (i = 0; i < size; i++)
    {
        UART1_SendByte(bmp[i]);
        for (k = 0; k <= 50; k++)  //delay
    }
    OS_EXIT_CRITICAL(); // Enable interrupts for critical section implementation
}
```
message. Once mailbox receive message from Task2, OSMboxPend() resume execution immediately after call OS_Sched(). Now Task2 and Task4 moved to ready state from blocked state. Task4 will execute first because Task4 has critical section which will block all tasks by disabling interrupt. Once Task4 is completed then Task2 will execute. The pseudo code for inter process communication using mail box is given below.

```c
void uctsk_error (void *data)
{
    int data = 0;
    int i;
    void* msg;
    INT8U err;
    while (1)
    {
        OSFlagPend(UserStatus,
            USER_INPUT_OCCURED,OS_FLAG_WAIT_SET_ALL+OS_FLAG_CONSUME, 0, &err);
        //Get status values to data
        EUARTputString("Enter uctsk_error \n");
        err = OSMboxPost(MBox1, (void*)&data);
    }
}
```

![Figure 6.18 Inter Processor Communication using Mailbox](image)
Pseudo code for mailbox implementation is given below.

```c
static void uctsk_zigbee (void *pdata)
{
    unsigned int i, j, size, k;
    uint8_t err = 0;
    UART0_SendString("\r\nTask 2(uctsk_zigbee) is Running\r\n");
    bmp = OSMboxPend(MBox, 100, &err);// Waiting for mail from mailbox
    UART1_SendString("\r\nZigBee DATA IS :");
    size = sizeof(bmp);
    for(i=0; i<size; i++)
    {
        UART1_SendByte(bmp[i]);
        for(k=0; k<=50; k++) //delay
    }
}
```

### 6.5 CHAPTER SUMMARY

The transformation for an existing non realtime liquid level control system is implemented using preemptive RTOS preemptive concepts. In multitasking RTOS system data corruption and intertask communication are addressed using \( \mu \)C/OS-II. The data’s are protected using critical section implementation in multitasking system. Inter task communication is implemented using mail box communication methodology.