**Serial Based Text Messaging System**

CE: 426 – Real Time Embedded Systems

Professor Foster

Fall 2010

Jacob Howarth, Wyatt Paro, Scott Snyder

Table of Contents

[1. Introduction 5](#_Toc280274371)

[2. Tasks 5](#_Toc280274372)

[2.1 AppStartTask 5](#_Toc280274373)

[2.1.1 Description 5](#_Toc280274374)

[2.1.2 Arguments 6](#_Toc280274375)

[2.1.3 Stack Size 6](#_Toc280274376)

[2.1.4 Priority 6](#_Toc280274377)

[2.2 LCD\_TestTask 6](#_Toc280274378)

[2.2.1 Description 6](#_Toc280274379)

[2.2.2 Arguments 7](#_Toc280274380)

[2.2.3 Stack Size 7](#_Toc280274381)

[2.2.4 Priority 7](#_Toc280274382)

[2.3 KeypadRdTask 7](#_Toc280274383)

[2.3.1 Description 7](#_Toc280274384)

[2.3.2 Arguments 8](#_Toc280274385)

[2.3.3 Stack Size 9](#_Toc280274386)

[2.3.4 Priority 9](#_Toc280274387)

[2.4 SPIWriteTask 9](#_Toc280274388)

[2.4.1 Description 9](#_Toc280274389)

[2.4.2 Arguments 9](#_Toc280274390)

[2.4.3 Stack Size 10](#_Toc280274391)

[2.4.4 Priority 10](#_Toc280274392)

[2.5 SevenSegTestTask 10](#_Toc280274393)

[2.5.1 Description 10](#_Toc280274394)

[2.5.2 Arguments 10](#_Toc280274395)

[2.5.3 Stack Size 11](#_Toc280274396)

[2.5.4 Priority 11](#_Toc280274397)

[2.6 SpeakerPlyTask 11](#_Toc280274398)

[2.6.1 Description 11](#_Toc280274399)

[2.6.2 Arguments 11](#_Toc280274400)

[2.6.3 Stack Size 12](#_Toc280274401)

[2.6.4 Priority 12](#_Toc280274402)

[2.7 SCI Interrupt Service Routine 12](#_Toc280274403)

[2.7.1 Description 12](#_Toc280274404)

[2.8 Other Useful Functions 13](#_Toc280274405)

[2.8.1 HardwareInit 13](#_Toc280274406)

[2.8.2 SemaInit 13](#_Toc280274407)

[2.8.3 MutexInit 14](#_Toc280274408)

[2.8.3 grabInfo 14](#_Toc280274409)

[2.9 MicroC/OS-II Modifications 14](#_Toc280274410)

[3. Events and Services 15](#_Toc280274411)

[3.1 Semaphores 15](#_Toc280274412)

[3.1.1 SCISemWrt 15](#_Toc280274413)

[3.1.2 SpeakerPlaySem 15](#_Toc280274414)

[3.2 Mutexes 15](#_Toc280274415)

[3.1.1 SPIMutex 15](#_Toc280274416)

[3.3 Message Queues 16](#_Toc280274417)

[3.4 Memory Partitions 16](#_Toc280274418)

[4. Data Structures 16](#_Toc280274419)

[4.1 Node Structure 16](#_Toc280274420)

[4.2 Free Block Linked List (Singly Linked List) 17](#_Toc280274421)

[4.3 In Use Block Linked List (Doubly Linked List) 17](#_Toc280274422)

[4.4 Linked List Functions 17](#_Toc280274423)

[4.4.1 Init\_list 17](#_Toc280274424)

[4.4.2 remove\_free\_list 17](#_Toc280274425)

[4.4.3 remove\_use\_list 18](#_Toc280274426)

[4.4.4 isEmptyFree 18](#_Toc280274427)

[4.4.5 use\_get\_end 18](#_Toc280274428)

[4.4.6 get\_head\_of\_used 18](#_Toc280274429)

[5. Assumptions 19](#_Toc280274430)

[5.1 No Error Handling 19](#_Toc280274431)

[5.2 Text Message Character Limit 20](#_Toc280274432)

[5.3 Number of Text Messages 20](#_Toc280274433)

[5.4 No Debouncing Mechanisms 20](#_Toc280274434)

[6. Lessons Learned 21](#_Toc280274435)

[6.1 Scott Snyder 21](#_Toc280274436)

[6.2 Wyatt Paro 21](#_Toc280274437)

[6.3 Jacob Howarth 22](#_Toc280274438)

# 

# 1. Introduction

This paper will discuss the details of the serial based text messaging system developed using the MicroC/OS-II real time operating system (RTOS) and the HCS12 microcontroller as the final project for this course. The paper will discuss the specific elements of system, including created tasks, used events and services, specific data structures, and interrupt service routines (ISR’s) implemented in the system. The paper will also discuss the basic assumptions of the system and its capabilities. The paper will conclude with a lessons learned section outlining the specific problems encountered during development and their solutions.

# 2. Tasks

This section will describe each task created and used in the project. Each task within the project will have a brief description of its function as well as a description of its elements including the priority, stack size, and arguments. Interrupt service routines will also be discussed along with any modifications done to the MicroC/OS-II operating system.

## 2.1 AppStartTask

### 2.1.1 Description

This task is used to create and initialize all other tasks within the project. Specifically, it calls the functions BSP\_Init(), OSStatInit(), AppProbeInit(), and AppTaskCreate(). The BSP\_Init() function initializes the ticker and other hardware functions such as the seven segment display, speaker, LCD, buttons, and other hardware components supported by the Dragon-12 board. The OSStatInit() is startup code used to calculate the maximum value a 32-bit counter can reach when no other task executes. It is called from the first created task and when multitasking is started. The AppProbeInit() is a function which initializes the MicroC/Probe features which allows monitoring of memory locations, variables, ports, and other facets of the operating system while it runs. These functions are mostly extraneous and involve initializing components related to the kernel which the programmer doesn’t interface with.

The AppTaskCreate() function is the function that creates all the tasks implemented in this project. The function creates the LCD\_TestTask(), SevenSegTestTask(), KeypadRdTask(), SpeakerPlyTask(), and SPIWriteTask(). The function also initializes all the task names, stacks, and priorities. Essentially, the AppStartTask is a make-shift idle task that creates all the tasks used in the project and then waits and does nothing after the tasks are created.

### 2.1.2 Arguments

The AppStartTask() has no arguments passed to its void \*p\_arg argument pointer.

### 2.1.3 Stack Size

The AppStartTask() has a local stack size of 256 bytes. This is the default size for this task.

### 2.1.4 Priority

The AppStartTask() is what creates all other tasks in the system. Therefore, this task should be given the highest priority. Since lower numbers indicate higher priority in MicroC/OS-II then a priority of 1 is assigned to this task.

## 2.2 LCD\_TestTask

### 2.2.1 Description

The LCD task was used for displaying the text message on the LCD display on the Dragon12+ board. We used some of the code that was already supplied by this task, and modified it to fit our application. After LCD initialization, we start off at the infinite loop. This loop first checks and makes sure that that there are messages in our used device linked list. If not, we will clear the LCD display using the provided functions. We then make sure that the current message were pointing at has more than one message in the device list. If not, we assign the current message pointer to the head of the in use linked list. This check is mostly used as an initialization technique, insuring that our current message is pointing to our used linked list.

Within this task, we use a variable which acts as a flag to make sure that we are operating on the correct message. This flag is initially at zero. Once a message arrives in our used linked list a conditional statement is used to check the flag, if the flag is at zero (which initially it is) it is passed to a function called grabInfo, which is explained in section 2.8.3.

After the information is obtained, the LCD will eventually cycle to the piece of code which takes care of displaying the lines from our LCD buffer to the LCD itself. We used a previously created function for these purposes. The display function will display whichever line we supply from our LCD buffer and we use a variable to access the lines of the text message which can be changed by using the keypad task to scroll through the message (if exceeding 32 characters). The LCD task will continuously display this message until we either delete the message or go to another message in the used linked list.

### 2.2.2 Arguments

The LCD\_TestTask() has no arguments passed to its void \*p\_arg argument pointer.

### 2.2.3 Stack Size

The LCD\_TestTask() has a local stack size of 600 bytes. This is the default size that was initially assigned for this task. There is no identifiable reason why it is larger than 256 bytes, but this was assigned to the base code.

### 2.2.4 Priority

The LCD\_TestTask() was assigned a value of 13 which is the lowest priority task of the system. Since MicroC/OS-II schedules higher priority tasks firsts, this task was assigned a priority lower than the SPIWriteTask, SevenSegTestTask, SpeakerPlyTask, and KeypadRdTask since all of these tasks perform operations that modify a part of the system and the LCD\_TestTask() just displays text messages stored in the SPI buffer and display tasks that only read are usually considered low priority tasks.

## 2.3 KeypadRdTask

### 2.3.1 Description

The keypad read task was used as an active polling mechanism for the buttons on the Dragon12+ board. We used the keypad task that was initially found in the project, and modified it to fit our needs. We kept a lot of the code, like the key mapping info, as well as the keypad init functions. Essentially, we added code to provide the following functions for our project:

* Scroll down through a text message if exceeding 32 characters, key # on the button array
* Scroll up through a text message, key 6 on the button array
* Scroll to the next text message, key C on the button array
* Scroll to the previous text message, key 8 on the button array
* Delete a text message, key 9 on the button array

**Scrolling Up/Down through a message:**

In order to scroll up and down through a message, we would first assign a key (two actually) from the keypad to some conditional statement. If it evaluated to true (key was pressed), we would provide additional functions. First, we would increment/decrement the LCD variables used to go through a message line by line. We would then clear the screen, and our LCD task would take care of displaying the new line(s) of text. For more information on how the LCD performs this function, please refer to the LCD task.

**Scrolling through messages:**

In order to scroll through messages, we would first assign a key (two actually) from the keypad to some conditional statement. If it evaluated to true (key was pressed), we would provide additional functionality. Essentially, we would just move the pointer that was pointing to the message being displayed on the LCD to the next/previous message (depending on which key was pressed). If there was only one message, nothing would happen, if there were multiple messages and we were at the last message in the list, we would wrap around to the beginning/end of the messaging queue (depending on which button was pressed) and set a flag to grab the correct information (check grabInfo function used within the LCD task).

**Deleting a message:**

In order to delete a message, we would first assign a key from the keypad to some conditional statement. If it evaluated to true (key was pressed), and there was a message that was in the in use linked list, we would provide the functions needed to delete the message. Essentially, we would take the pointer to the message, change it, and then remove the message by passing the base that was contained in the node, to a remove function. This function would take care of removing the message from the in use list and putting it back onto the free list. The pointer to this removed message will either be moved to the next message in the list, to the beginning of the messaging queue, or nulled (no messages). The LCD will update accordingly after this has been completed.

### 2.3.2 Arguments

The KeypadRdTask() has no arguments passed to its void \*p\_arg argument pointer.

### 2.3.3 Stack Size

The KeypadRdTask() has a local stack size of 256 bytes. This is the default size for this task.

### 2.3.4 Priority

The KeypadRdTask() was assigned a priority of 6 as its default value. This was the initial assignment to the unmodified base code in textMsg.zip given at the start of the project.

## 2.4 SPIWriteTask

### 2.4.1 Description

This task was created to write the inputted data to the appropriate address into the 256K SPI bus low-power serial SRAM. At task creation, we created a semaphore which would pend in this task before proceeding to write any information to the SRAM. The SCI interrupt handler would be in charge of posting to this semaphore. The SCI Handler would post to the semaphore either after the total amount of characters reached 160, or if the user pressed the enter key (for the text message). After the post, the task would then grab the last block on the used linked list, which contained the address to where we wanted to write to for the SPI SRAM, pass that address to a SPI RAM write function along with the message.

This function would then split the address supplied into both an upper byte and lower byte using a union. Send the appropriate command to the SPI device, send the address, and finally send the message until a null character was reached. Once reached, the SPI would write the null character to SRAM and complete. The SPIWriteTask would then go back to pending on the semaphore until the user supplies another message.

### 2.4.2 Arguments

The SPIWriteTask() has no arguments passed to its void \*p\_arg argument pointer.

### 2.4.3 Stack Size

The SPIWriteTask() has a local stack size of 256 bytes. This is the default size for this task and was randomly assigned.

### 2.4.4 Priority

The SPIWriteTask() has a relatively high priority of 2 since it deals with writing to the SPI buffer which is a critical section of code. It is also controlled by the SCI ISR because the task blocks until the SCI ISR posts to the SCISemWrt semaphore.

## 2.5 SevenSegTestTask

### 2.5.1 Description

This task controls the 4 seven segment displays on Port B of the HCS12 on the Dragon-12 board. Specifically, this task updates the clock of the text message system using a polling method. The seven segment displays are first initialized using the SevenSegDisp\_Init() function and then the display is set to 12 hours, 0 minutes, and 0 seconds using the SevenSegWrite(int) function which takes an integer as an argument. Both the SevenSegDisp\_Init() and the SevenSegWrite(int) functions are in the sevenSegment.c file which contains driver code for the seven segment displays.

The current time for the clock is displayed based on a base 10 system, meaning 12:00 is equivalent to the integer 1200 being written to the seven segment display with the SevenSegWrite (int) function. Since the clock doesn’t keep track of second’s, a second’s counter, called centisecondCounter is used as a modulus counter that’s incremented after each write to the seven segment display. If the centisecondCounter mod 6000 is equivalent to 0, the minute’s portion of the clock is updated. The minute’s value is then checked to see if it is greater than 59, causing an update to hours and a reset of the minutes to zero. The clock operates in a 12 hour mode, meaning once the hours counter becomes greater than 12, it wraps around to 1. A delay is used to prevent this from happening continuously.

### 2.5.2 Arguments

The SevenSegTestTask() has no arguments passed to its void \*p\_arg argument pointer.

### 2.5.3 Stack Size

The SevenSegTestTask() has a local stack size of 256 bytes. This is the default size for all the tasks.

### 2.5.4 Priority

The SevenSegTestTask() is used for updating and displaying the text message system clock. The priority of the task was assigned a value of 4 since it is a fairly high priority task, but not as high as any of the data gathering and writing functions.

## 2.6 SpeakerPlyTask

### 2.6.1 Description

This task plays the ringtone on the speaker component of the Dragon-12 board whenever a text message is received through the serial port from a terminal program. It is initiated on the pressing of the Enter key on the keyboard. The Enter key, when pressed in the terminal program, sends a carriage return character (i.e. “\r\n” in Windows) to the SCI which causes the SCI ISR to post to a semaphore which plays the ringtone. The ringtone uses the sound driver in speaker.c file to play tones signifying the individual notes in different octaves found in the speaker.h file. The ringtone can also be activated by pressing the button labeled SW5 on the Dragon-12 board which is controlled by Port H on the HCS12. This button was used primarily for testing purposes.

The ringtone plays “Crazy Train” by Ozzy Osbourne by instituting delays of various times between each note to simulate whole notes, quarter notes, half-notes, dotted half-notes, and eighth notes.

### 2.6.2 Arguments

The SpeakerPlyTask() has no arguments passed to its void \*p\_arg argument pointer.

### 2.6.3 Stack Size

The SpeakerPlyTask() has a local stack size of 256 bytes. This is the default size for all the tasks.

### 2.6.4 Priority

The SpeakerPlyTask() is used for playing the ringtone upon . The priority of the task was considered to be low because it does very little and does not affect any type of text message data in terms of writing and reading. Thus, the priority was assigned a value of 11.

## 2.7 SCI Interrupt Service Routine

### 2.7.1 Description

The SCI handler was used as an interrupt subroutine for handling data input via the keyboard. This input would eventually be stored into a buffer, which was our text message, and stored for later use. The SCI handler would be called whenever the user started typing a message on the keyboard. The characters would be counted and put into a global buffer. Once the amount of characters reached 160 or if the user pressed the enter key, the handler would provide the following functions:

* Append a null character at the end of the buffer (our text message)
* As long as there are nodes in the free linked list, we will remove a node from the linked list and add it to the in use linked list, for more details please refer to section four.
  + If the free list is empty, that means that we cannot hold anymore messages, so buffer is then erased and we do not proceed.
* Post the semaphore for the speaker task, allowing the ringtone to chime
* Post the semaphore for the SPIWriteTask allowing the message that is contained in the global buffer to be written to the SPI SRAM.

After the above actions are completed the counter is set to zero. The global buffer will be erased in the SPIWriteTask.

## 2.8 Other Useful Functions

### 2.8.1 HardwareInit

This function was used to initialize all the hardware used in the text message system. It is called in main() after OSInit() and initializes and configures the following hardware elements:

* Init\_list() – initializes the free linked list and use linked list for the SPI memory management functionality. This can be found in the spi\_mem.c file.
* Switches\_Init() – Initializes the buttons labeled SW2, SW3, and SW5 on the Dragon-12 board used for incrementing of the hour and minute scales on the clock and playing the ringtone respectively. Specfically, it writes 0x00 to the data direction register for port H (DDRH) signifying inputs and sets the local interrupt enable flags on port H (PHIE) to 1 for SW2, SW3, and SW5. It also initializes the semaphores for each switch. However, these values are never used since the buttons are not interrupt driven. This can be found in the switches.c file.
* Sci1\_Init() – Initializes and configures the serial communications interface one (SCI1) for receiving text messages sent from the terminal program on the PC. Specifically, it sets the baud rate, data bits, parity, stop bits, and receiver (Rx) and transmitter (Tx) interrupts. This can be found in the Sci\_c.c file.
* spi0\_init() – Initializes and configures the serial peripheral interface (SPI) buffer used to store the text messages sent from the terminal program on the PC. Specifically, it initializes and configures the baud rate, interrupt, chip select, and RAM of the SPI device and uses SCI0 to write and read from it. This can be found in the spi.c file.
* Speaker\_Init() – Initializes and configures the speaker on the Dragon-12 board. Specifically, it initializes and configures the PWM5 register and its control register PWMCTL by setting the appropriate clocks, prescalars, and interrupt flags. This can be found in the speaker.c file.
* SemaInit() and MutexInit() – Creates and initializes the semaphores and mutexes used in the tasks and interrupts for the project. These functions will be explained in sections 2.8.2 and 2.8.3. They can be found in the app.c file.

This function takes no arguments and returns no value.

### 2.8.2 SemaInit

This function creates and initializes the following semaphores used in the project:

* SpeakerPlaySem – A binary semaphore used to block the SpeakerPlyTask.
* SCISemWrt – A binary semaphore used to block the SPIWriteTask.

More details on the use of these two semaphores and what they control can be found in section 3 Events and Services.

### 2.8.3 MutexInit

This function creates and initializes the following mutex used in the project:

* SPIMutex– A mutex that protects the writing of the SPI buffer in the SPIWriteTask.

More details on the use of these two semaphores and what they control can be found in section 3 Events and Services.

### 2.8.3 grabInfo

This function will first clear our LCD display buffer (making sure that our message doesn’t display extra characters). Next we access the base of the node that our current LCD message pointer is pointing to. This base is the address in the SPI SRAM where we stored our text message. We pass this base along with the LCD display buffer to our SPI SRAM read function. This function will put the message that we stored in SRAM into our LCD display buffer. We then reset the rows in which our LCD task is using to display information so we can begin at the beginning of the message. After doing so we set the flag to one, ensuring that we do not access the grabInfo function until we need to go to another message (used in the keypad task, refer to this task).

## 2.9 MicroC/OS-II Modifications

No modifications were made to the MicroC/OS-II source code in this project.

# 3. Events and Services

This section will describe the MicroC/OS-II events and services used in the project. It describes in detail the use of any semaphores, mutexes, message queues, and memory partitions used within the project.

## 3.1 Semaphores

### 3.1.1 SCISemWrt

This is a binary semaphore that blocks the SPIWriteTask from executing. It is initially assigned a value of zero and the SCI ISR posts to this value after it reads in a text message from the terminal with a carriage return. After the post, it allows the SPIWriteTask to write the text message to the SPI buffer. It is created and initialized to zero in the SemaInit() function in the app.c file. It is posted to in the Sci\_c.c file in the SCI\_ISR\_Handler(). The semaphore will wait forever on a pend.

### 3.1.2 SpeakerPlaySem

This is a binary semaphore that blocks the SpeakerPlyTask from executing. It is initially assigned a value of zero and the SCI ISR posts to this value after it reads in a text message from the terminal with a carriage return. After the post, it plays the ringtone signaling a new text message. It is created and initialized to zero in the SemaInit() function in the app.c file. It is posted to in the Sci\_c.c file in the SCI\_ISR\_Handler(). The semaphore will wait forever on a pend.

## 3.2 Mutexes

### 3.1.1 SPIMutex

This mutex is used to establish a critical section in the SPIWriteTask where the text message is written to the SPI buffer at a specified address. This mutex is to insure that the operation is completed without any interference from contending read operations. The mutex was assigned a priority of 12 which is one above the SPIWriteTask priority of 11. This priority was assigned largely at random because of an initial design decision that we later abandoned. However, the mutex only operates within the SPIWriteTask and no other task uses the same mutex so priority inversion won’t occur. Therefore, the priority assigned to the mutex used to promote the tasks priority in case a higher priority task holds the mutex is not important.

## 3.3 Message Queues

No message queues were used in this project.

## 3.4 Memory Partitions

No memory partitions were used in this project.

# 4. Data Structures

This section describes the data structure used to manage the SPI buffer for inserting and deleting text messages. The subsections will describe the node structure, the linked lists used to manage the free and in use blocks of memory in the SPI buffer, and the functions used to manage the lists.

## 4.1 Node Structure

The structure of a node used in both the free linked list and the in use linked list consists of the following properties:

* Int base – This value holds the base address or starting address of where a text message will be stored in the SPI buffer.
* A previous node pointer – This pointer contains the address of the previous node in the in use linked list. This pointer is not used in the free linked list.
* A next node pointer - This pointer contains the address of the nextnode in the in use linked list and free linked list.

## 4.2 Free Block Linked List (Singly Linked List)

The free linked list is a singly linked list used to hold nodes which represent text messages that have not been used yet. We dedicate 20 nodes onto the linked list at initialization. When a message is written and submitted by the user, a node is taken off of the free linked list and put into the in use linked list. When we reach 20 messages, the user will not be able to add anymore messages because the free list will be empty. More information on how nodes are added to the free list can be found in both the initialization and remove from used list functions. Nodes that are removed from the free list can be described in additional detail within the remove from free function.

## 4.3 In Use Block Linked List (Doubly Linked List)

The in use linked list is used to hold the nodes that contain the addresses of where to access the text message information from the SPI SRAM. This list will be empty (NULL) at initialization. When a text message is entered by the user, a node will be removed from the free list and put into the in use list. The in use list is a doubly linked list. The list was made this way so it would be easier to go through the text messages and display them on the LCD. Adding to the in use list can be referenced in the remove from free list function. Deleting from the in use list can be found in the remove from used list function.

## 4.4 Linked List Functions

### 4.4.1 Init\_list

This function is used to populate the free list with nodes. Each node that is added to the free linked list at initialization is initialized with a base address, which is an address which will be used when accessing the SPI SRAM. When we add the first node, we start the base address at zero. Upon adding additional nodes, we add an offset of 160. This offset is used because we will be writing messages to the SPI SRAM that are at most, 160 characters. For example, node one will have a base of 0 and be able to write up to 159 in the SRAM, node two will have a base of 160 and write to 319, node 3 base of 320 etc… This function creates a total of 20 nodes, i.e. 20 text messages that can be created.

### 4.4.2 remove\_free\_list

This function is used to remove a node from the free linked list and add it to the in use linked list. When removing from the free list, we always remove the first node that the head is pointing to. If the head is null, then we know that the free list is empty, so we will not proceed with this function. If not, the first node is grabbed and the free list head pointer is reassigned to the next node. Next the node is sent to the end of the in use linked list. This process will first check to see if the in use linked list head is null, if so it just assigns the head to this node. Otherwise, the in use linked list is traversed all the way to the end and added to the end. The previous pointer of the node we added is then initialized to the node that is pointing to it. The function then exits.

### 4.4.3 remove\_use\_list

The remove use list function is in charge of removing a particular node from the in use linked list. The supplied argument is used in order to delete the appropriate node from the list. The value is the base address, or the address that is used when writing to the SPI SRAM. We start by assigning a node pointer to the head of the in use list. We then traverse the list, checking each node’s base value until we reached the node that matches the argument. When we reach this node, we assign the nodes previous node to the current nodes next pointer and the node that the current node that we are removing previous pointer to the one before the current node. Now that we removed the node from the list, we null both the nodes next and previous pointers. Next, we add the node that we removed from the in use linked list, to the end of the free linked list. We do this by traversing the free linked list to the end, and assigning the next pointer of the last node on this list, to the node that we removed from the in use list.

### 4.4.4 isEmptyFree

This function checks if the free linked list is empty. If it is (head of free list is NULL) then we return a zero, otherwise we return a one (not empty).

### 4.4.5 use\_get\_end

This function is used to get the last node in the in use list. This function will traverse the list to the last node in the in use list and assign it to our global end node pointer.

### 4.4.6 get\_head\_of\_used

This function assigns the LCD message pointer to the beginning of the in use linked list head pointer.

# 5. Assumptions

This section will outline any assumptions on the design and operation of the text messaging system. There are four key assumptions that can be identified in the system:

1. No error handling is attempted on the semaphores, mutexes, and tasks.
2. The text messages can contain a maximum of 160 characters.
3. The range for the number of text messages allowed by the system is [1, 200], with an initial value at 20.
4. No debouncing mechanisms.

The section will also explain reasons for the assumptions and any improvements that can be made on the assumptions.

## 5.1 No Error Handling

Errors that occur with semaphores and mutexes are checked on their pends by passing an integer pointer to the OSSemPend() and OSMutexPend() functions. This occurs in the SpeakerPlyTask and the SPIWriteTask. The value of these integer pointers, after calls to the pends are made, are not checked within the program.

No error checking was made on the pend calls for the following reasons:

* The text message system falls into the beta category of development.
* The text messaging system is not production code and therefore doesn’t require extraneous error checking.
* There wasn’t enough time to write all the error checking in because it is largely redundant and often never gets executed within the program.

Error’s largely went unchecked because of the time crunch near the project due date. We also assumed that all semaphore and mutex pends would return with no errors and chose not to include error checking because it would require extra time that we didn’t have. Improving the project management side would have resolved this issue. This is discussed more in section 6 Lessons Learned section.

## 5.2 Text Message Character Limit

## 

The maximum number of characters a text message can have is 160 including the null character. The null character changes the total to 159 characters because the 159 position of the array contains a character with a value of 0. This constraint is due to the spi\_RAM\_write\_string() in the SPI driver located in spi.c. The function imposes a maximum limit of 160 characters that can be written to the specified address argument to the function.

## 5.3 Number of Text Messages

The number of text messages for the current system is 20. However, the system can theoretically have a number of allowable text messages in the range of [1, 200]. This constraint is largely due to the size of the SPI buffer and the limitations of the spi\_RAM\_write\_string() function in the spi.c file which contains the driver for the SPI buffer. The spi\_RAM\_write\_string() function wrote a string of 160 chars, including the null character, to the SPI buffer at a specified address. Since we wrote this function in lab 3 and it worked properly, a design decision was made to keep the function as is and break up the SPI memory into fixed blocks with each block containing a maximum of 160 characters. With the SPI buffer being a size of 256 Kbits, a calculation was used to determine the number of blocks the SPI memory could be divided into with this 160 character limitation.

Therefore, our linked list of free memory blocks used to manage the SPI buffer has a maximum of 200 blocks.

## 5.4 No Debouncing Mechanisms

No debouncing schemes were implemented on any of the buttons used in the project. The buttons with required for the project include SW2 used to increment hour on the clock, SW3 used to increment the minute on the clock, SW5 used to play the ringtone, and the keypad buttons 8, 9, C, 6, and # used to go to the previous message, delete a message, go to the next message, go to a previous line in a multi-line message, and advance to the next line in a multi-line message respectively. No of these buttons had debouncing mechanisms which resulted in poor performance when scrolling through the messages displayed on the LCD.

The main reason this wasn’t accomplished was because it wasn’t a requirement for the project. The exact place where the debouncing occurred in the code was largely unknown so there was no way of knowing where to implement the debounce scheme for the buttons due to debouncing being caused by hardware. An attempted solution was to prevent repeated post to semaphores caused by the button noise by decrementing the event control block (ECB) event counter down to 1 on every button press so the posts would be reduced to one on each button press. However, this was eventually abandoned for pulling methods for the buttons. The pulling method improved the performance of the SW2, SW3, and SW5 buttons, but the same issues emerged in the keypad buttons.

# 6. Lessons Learned

The following section describes the lessons learned from developing the text message system. The section is divided in to three subsections with each group member contributing an encountered obstacle within the project and the proposed solution or non-solution if one wasn’t able to be found.

## 6.1 Scott Snyder

While writing the SpeakerPlayTask, I was having trouble getting the notes to sound like the actual song. It turns out that I was an octave off. Once I adjusted this, the song sounded much better. However, the first part of our ringtone has two of the same notes in a row. When played, they sound like one really long note. After messing with the timing, it still did not work properly. I learned that I had to use a NOTE\_OFF in order to properly hear both of the individual tones. Once I added this in between, the ringtone sounded very much like our intended song, Crazy Train.

## 6.2 Wyatt Paro

During the implementation of both our in use and free linked lists, we came across some unusual behavior when modifying certain nodes. We found out that there was an issue when assigning a local node pointer to a node, and passing that local pointer to a function. The contents of the node that the pointer was pointing to would have random data stored in the node. We believe that this is because we are accessing or modifying information from other functions stacks. In order to fix this problem, we combined functions together and made global pointers. When we did this we did not experience the same issues. This problem consumed most of our time on the project.

Another bizarre issue that we encountered was upon linked list initialization (filling the free linked list with nodes) it would reset our board entirely. During debugging, when stepping through the code, once we reached the initialization function, the Dragon12+ board would reset and our project would obviously fail. After spending multiple ours debugging this problem, we came to the conclusion that we needed to implement a different data structure. The next day when we came into work on the project, we tried the linked list data structure again, and it *magically* worked. All that we did to change this is make our array of nodes global. We’re not entirely sure what made the board reset before we did this. But this seemed to fix the problem.

## 6.3 Jacob Howarth

One obstacle that occurred in the project was the debouncing problem. When the buttons SW2, SW3, and SW5 were implemented using ISR for port H, the clock would increment the hour or the minute multiple times on one press of SW2 and SW3 respectively. An attempted cause for the debouncing was the repeated posting to the semaphores for the buttons SW2, SW3, and SW5 caused by the noise from the button contacts. A proposed solution was to decrement the event control block (ECB) event counter for each semaphore down to 1 on each press of SW2, SW3, and SW5. However, this solution did not resolve the problem and the interrupt driven SW2, SW3, and SW5 buttons were abandoned for a pulling version which operated more cleanly. The exact same problem occurred for the keypad buttons used to manipulate the text messages displayed on the LCD. However, there wasn’t enough time to figure out a solution to this problem before the project was submitted.

A proposed cause to why the solutions failed was the needed call to the OSTimeDlyHMSM() on each button press before decrementing the semaphore event counter. This delay probably would have normalized the value of Port H for SW2, SW3, and SW5 based on how long a finger was pressed on the button.

Another obstacle was the lack of project management. The project was implemented with no schedule and no deadlines for the specific phases of the project. The project phases were not efficiently divided up for each partner in the group as well. A lack of these elements resulted in a hurried development process before the project due date which degraded group moral and performance. Dividing the project into phases and setting a schedule with deadlines for each phase offers more project stability, motivation for group members to do their tasks, and reduction in unnecessary stress and confusion resulting from unplanned work.