

**ECED 4402**

Real Time Systems

Electrical Engineering Department

**Assignment #2**

Lightweight Messaging Kernel

Written by:

Abdulrahman Aljedaibi

B00656986

Abdullah Alhadlaq

B00623068

Submission date:

15 Nov, 2017

Table of Contents

[1. Problem Definition 1](#_Toc494720665)

[2. Problem Analysis 2](#_Toc494720666)

[3. Solution Design 5](#_Toc494720667)

[3.1 UART module 5](#_Toc494720668)

[3.2 SYSTICK module 6](#_Toc494720669)

[3.3 Queue Module 6](#_Toc494720670)

[3.3.1 Queue Initialization 7](#_Toc494720671)

[3.3.2 Enqueuing 8](#_Toc494720672)

[3.3.3 Dequeuing 8](#_Toc494720673)

[3.4 Control Module 9](#_Toc494720674)

[3.4.1 Supporting Algorithms 11](#_Toc494720675)

[3.4.2 Command algorithms 12](#_Toc494720676)

[4. Data Dictionary 14](#_Toc494720677)

**Table of Figures**

Figure 1: Communication between user and microcontroller 2

Figure 2: Simple visualization of the inner workings of the simple monitor system. 3

Figure 3: Context Level Diagram of the simple monitor system. 3

Figure 4: Zero level diagram of the simple monitor system. 4

Figure 5: state diagram for UART interrupt service routine that is responsible for transmission. 5

Figure 6: Data structure of the input and output queues 7

Figure 7: state diagram of the machine state used in the control module. 9

Figure 8: state diagram for the character printing algorithm. 12

# Problem Definition

In this assignment, a simple monitor system is to be deigned, programmed, and tested. The system supports three commands. The “TIME” command allows the user to check the time of day by displaying the time of day on the monitor. The “SET” command allows the user to alter the current time of day stored in the machine. Finally, the “GO” command starts a stop watch with an accuracy of one-tenth of a second which can be stopped by sending any character. The system will be implemented on the TIVA microcontroller. The user can access the system by connecting an external computer with a PuTTY terminal opened to the TIVA through a USB cable. The user is to communicate with the monitor system through the PuTTY terminal.

This problem is divided into four major modules: UART module, SYSTICK module, queuing module, and a control module. This report will approach the problem by first giving a brief analysis of the problem as well as going over the four modules (section 2). Then, the design of each module will be explored individually in more detail (section 3). After that, a data dictionary will be provided containing the definitions and contents of the elements used in the system (section 4). Next, a C code implementation of the problem (section 5) will be given followed by a test suite to validate the behavior of the system (section 6).

# Problem Analysis

In this section, a general view of the system will be given. First, the analysis portion will cover the parameter of the problem and define what the solution need to accomplish. Then, the solution approach section will cover briefly the four major modules in the solution and explain how they all work to yield the monitor system.

The system needs to process three different commands which are:

* “TIME”
* “SET”
* “GO”

The commands are supposed to display the time of day, set the time of day, and start a stop watch, respectively. Also, the screen needs to scroll when the cursor reaches the bottom of the screen. Processing the commands and keeping track of the cursor position are objectives of the monitor system. In order to receive the commands from the user, TIVA needs to use the UART device to be able to communicate with the user. The user can send characters through the PuTTY terminal and TIVA can send characters to the user using the UART device as shown in Figure 1.



Figure : Communication between user and microcontroller

It is clear that the system needs to keep track of time since all the commands require manipulation and/or display of time. Therefore, the use of TIVA’s system clock is mandatory. In order to process the commands, the SYSTICK exception within the TIVA will be used. Initialization of SYSTICK as well as the algorithms used for handling the time interrupts we be discussed.

The system will be receiving clock interrupts every one-tenth of a second as well as characters from the user. The system might not be fast enough to process the inputs before another input comes in. Therefore, a queue module is required. An input queue is to be designed in order to store the incoming information pending to be processed by the system. Also, it should be noted that the system operates much faster than the serial communication speed at which the TIVA and the user are using. Therefore, an output queue is necessary to store the characters to be sent out to the user to give the UART enough time to send them.

In order to process the commands sent by the user, there needs to be a centralized module that processes the incoming characters and clock interrupts. The control module is responsible for such task. It will be dequeuing entries from the input queue and processing each one. It should be able to update the time of day and the stop watch accordingly. It should also be able to process the three commands as well as keep track of the curser position. To simply visualize how the four modules work together, Figure 2 is created.



Figure : Simple visualization of the inner workings of the simple monitor system.

Figure 2 does not show all the elements of the system. Therefore, a data flow diagram is a better technical tool to describe the system. Figure 3 below is context level diagram showing a high-level perspective of the system.



Figure : Context Level Diagram of the simple monitor system.

In the following diagram shown in Figure 4, the simple monitor system is expanded into the four components and they are displayed in a data flow diagram.



Figure : Zero level diagram of the simple monitor system.

# Solution Design

In this section, a detailed discussion of the Simple Monitor System’s algorithms and data structures. The four modules of the solution which are: UART, SYSTICK, queue, and control modules will be explained individually.

## Process registration

The initialization of the UART module is already discussed in great detail in the course website for ECED 4402. However, the interrupt service routine needs to be designed. IF the reception interrupt is triggered, then the ISR should enqueue the incoming character in the input queue using the algorithms explained in the Queue module section. However, if the transmission interrupt is triggered, then state diagram in Figure 5 should be followed. It should be noted that states in the diagram are for the UART\_state.



Figure : state diagram for UART interrupt service routine that is responsible for transmission.

Also, this algorithm can be explained using structured English as follows.

UART Interrupt Service Routine algorithm

IF it is a receive interrupt THEN

Clear receive interrupt flag

Enqueue the character in the input queue

ELSEIF it is a transmit interrupt THEN

Clear transmit interrupt flag

IF the output queue is not empty THEN

Dequeue a character from the output queue

Output the character

ELSE

Set UART\_state to IDLE

ENDIF

ENDIF

## SVC and process initiation

The initialization algorithms for the SYSTICK module is discussed in great detail in the course website. However, the interrupt service routine needs to be designed. One main issue that needs to be addresses is that there is no fixed period that will produce one-tenth of a second. The machine maximum number of ticks is 2^24 = 16,777,216. So, we can tell the machine to notify us every 1,677,721 ticks and assume that elapsed time is equal to one-tenth of a second. However, one-tenth of a second is actually 1,677,721.6 ticks long. This means that the time in the machine will be drifting away from the actual time.

To resolve time drifting, we note that 2^24 ticks will cause one second to elapse. So, if we make the machine cause an interrupt every 2^14 ticks, then a second will have elapsed after 2^10 ticks. This means that one-tenth of a second will elapse every 102.4 ticks. Using this, we can tell the interrupt service routine to count the following numbers before enqueuing into the input queue that one-tenth of a second has elapsed:

Time adjust list = 102, 102, 103, 102, 103 ticks.

So, after 102 ticks enqueue a SYSTICK, then count to 102 again and enqueue, then counter to 103 and enqueue, and so on until the end of the previous list. Then you can loop back through the list again.

The interrupt service routine algorithm is described below in structured English format keeping in mind the time drifting issue.

SYSTICK Interrupt Service Routine algorithm

Increment the elapsed ticks counter

IF elapsed ticks counter = entry from the time adjust list indexed by the time adjust index THEN

Increment the time adjust index

Reset the elapsed ticks counter

Enqueue a SYSTICK entry into the input queue

ENDIF

## SYSTICK and PENDSV

The queue module is responsible for storing information temporarily until the dedicated section of the system has time to process it. Mainly, there are two queues: on for input, and one for output. The input queue will take entries from the UART module as well as the SYSTICK module. The output queue will take entries from the control module. Each queue entry should contain:

* Type (SYSTICK, or UART)
* Character (if it is UART)

If the entry is from SYSTICK, then it will always represent an elapsed time of one-tenth of a second so there is no need for time information to be stored in the entry.

A linked list was considered when designing the queue module. However, this is a real time system which means that time is the most pressing constraint. Linked lists are good for conserving space and they have the ability to grow and shrink in size dynamically. Unfortunately, they also have long overheads when adding or removing entries which is why they were not chosen for this application. Instead, an array will be used. Arrays are static but entries can be added and removed easily and quickly. Also, since we will not be needing to removing entries from the middle of the queue, using an array becomes a very attractive approach.

The input queue is an array consisting of 32 entries containing (TYPE, and CHAR). The queue will also contain three integers representing the head, the tail, and the counter. The head is a number that represents the position of the entry at the top of the queue, and so does the tail. The counter will keep track of the number of entries in the queue. The queue data structure looks as shown in Figure 6.



Figure : Data structure of the input and output queues

The output queue will have the same data structure with a size of 32. This size was chosen because the UART operates at a much lower rate than the system clock so the queue needs to be bigger in size to give the UART enough time process the characters.

### Queue Initialization

The Algorithm for initializing the queues are very simple. The following steps should be followed:

* Initialize the head to have a value of 0.
* Initialize the tail to have a value of 0.
* Initialize the queue\_counter to have a value of 0.

### Enqueuing

To add an entry to the head of a queue, the enqueuing algorithm must be followed. First, the queue is check if it is full by checking the queue\_counter and making sure it is less than the maximum queue size (which is 32). Then if the queue is not full, the entry is filled with the information to be added. Next, the queue\_counter is incremented and the head of the queue is updated. This algorithm is shown below in a structed English format.

Enqueue Algorithm:

*IF queue\_counter = MAX\_ ENTRIES (which is 32) meaning that it is full THEN*

*Exit and return an error code (specified by a value of 0)*

*ELSE*

*Add entry to the head of the queue*

*Update the position of the head of the queue (by incrementing the head integer)*

*Increment the queue\_counter*

*Signal successful enqueuing by sending a value of 1 to the calling subroutine*

*ENDIF*

*DONE*

### Dequeuing

To dequeue an entry from the tail of a queue, the queue is first checked to see if it is empty by checking the queue\_counter. If it is not empty, then supply the entry information to the calling subroutine. Next, update the position of tail and decrement the queue\_counter. This is shown below in a structured English format.

Dequeue Algorithm:

*IF queue\_counter = 0 meaning that it is empty THEN*

*Exit and return an error code (specified by a value of 0)*

*ELSE*

*Supply entry information (TYPE, and CHAR) to the calling subroutine*

*Update the position of the tail of the queue (by decrementing the tail integer)*

*decrement the queue\_counter*

*Signal successful dequeuing by sending a value of 1 to the calling subroutine*

*ENDIF*

*DONE*

## System commands

This module is responsible for processing the incoming characters and system clock interrupts. This is done by first dequeuing the entries from the input queue. Therefore, the control module is pulling the input queue until it is able to dequeue an entry.

Characters and time interrupts will be processed differently if the machine is in stop watch mode or not. So, the machine will have two states: stop watch state (SW), and not stop watch (NS). In the stop watch made, an incoming character will signal the end of the stop watch mode. Also, a time interrupt will have to update both the time of day and the stop watch time. If the machine is the NS state, then a character will be buffered into an input string and a time interrupt will update the time of day only. This is displayed in Figure 7 below in the form of a state diagram.



Figure : state diagram of the machine state used in the control module.

The operation of the control module can be better explained using structed English format as shown below. The module is an infinite loop and the algorithm below is one cycle of it. Note that the underlined words in the structured English algorithm are other algorithms that will be explored later on in this section:

Control Algorithm:

*Dequeue an entry from the input queue*

*IF the machine state = NS THEN*

*IF the entry is from SYSTICK THEN*

*Update the time of day by incrementing the time of day counter*

*ELSEIF the entry is from UART THEN*

*Process the character*

*ENDIF*

*ELSEIF the machine state = SW THEN*

*IF the entry is from SYSTICK THEN*

*Update the time of day by incrementing the time of day counter*

*Update the stop watch by incrementing the stop watch counter*

*Display the stop watch*

*ELSEIF the entry is from UART THEN*

*Machine state = NS*

*Go to the next line on the monitor*

*Echo back the same input character*

*Reset the stopwatch*

*ENDIF*

*ENDIF*

The process character algorithm underlined in the previous structured English algorithm will be discussed. When the machine is the NS state, the incoming characters must be processed. There is an input buffer that will contain the incoming characters and it has a size of 128 characters (BUFFER\_SIZE). If we receive a character, then we need to echo it back and append it to the input buffer. If a carriage return character is received, then an end of line character is sent and the input buffer is null terminated and is passed to the algorithm responsible for checking the incoming string. If instead we receive a delete character, then a character should be deleted from the input buffer and it is echoed back to the user. In the case that the input buffer is full, the system should ignore the incoming characters. This is explained in the following structured English algorithm.

Process character algorithm:

*Echo back the received character to the user*

*IF the incoming character is a carriage return character THEN*

*Echo back a new line character to the user*

*IF the input buffer is not full THEN*

*Null terminate the input buffer*

*ELSE*

*Overwrite the last character and put a null character*

*ENDIF*

*Send the string to the check command process*

*Clear the input buffer*

*ELSEIF it is a delete character THEN*

*IF the input buffer is not empty THEN*

*Remove the last character from the input buffer*

*ENDIF*

*ELSE*

*IF the input buffer is not full THEN*

*Add the character to the input buffer*

*ENDIF*

*ENDIF*

After processing the incoming characters and forming a string that contains the command, the string must be compared with three commands: “TIME”, “SET”, “GO”. This can be done by parsing the string until we find a space or a null character. Then, compare the first token of the string to the three commands. If there is a match, then the corresponding command algorithm is executed. If there is not match, a ‘?’ will be sent to the user signaling an unidentified command. This process is explained below in structured English.

Check command algorithm:

*Parse the incoming string with the delimiters: space character and null character and get the first token*

*Transform the first token to upper case letter*

*IF the first token = “TIME” THEN*

*Execute the time algorithm*

*ELSEIF the first token = “GO” THEN*

*Execute the go algorithm*

*ELSEIF the first token = “SET” THEN*

*Execute the set algorithm*

*ELSE*

*Print ‘?’ to the user and go to the next line*

*ENDIF*

It should be noted that the second token should be checked to make sure that the “TIME” and “GO” commands do not have second tokens. Also, the third token should be checked to make sure that the “SET” command does not have a third token. This will be done in each command algorithm individually.

### Supporting Algorithms

Before discussing the command algorithms, there are some supporting routines that are used by the control module which should be explained. The time of day is kept by means of a counter; whenever there is a SYSTICK, the counter is incremented. This is also done with the stop watch. To be able to display this information, the hours, minutes, seconds, and tenths-of-seconds have to be extracted from the counter. The character ‘0’ has the ASCII number 48. The remaining nine Arabic numerals follow contiguously. So a single digit number can be transformed to its corresponding character by adding a value of 48. The following algorithm explains how to extract the time from the counter.

Get time algorithm

/\* the ‘%’ sign denotes the modulus operation \*/

/\* CHAR\_INDX = 48 \*/

*Tenths-of-second section = (counter % 10) + CHAR\_INDX*

*Ones digit of the seconds = [(counter / 10) % 60] % 10 + CHAR\_INDX*

*Tens digit of the seconds = [(counter / 10) % 60] / 10 + CHAR\_INDX*

*Ones digit of the minutes = [(counter / 600) % 60] % 10 + CHAR\_INDX*

*Tens digit of the minutes = [(counter / 600) % 60] / 10 + CHAR\_INDX*

*Ones digit of the hours = [(counter / 36000) % 24] % 10 + CHAR\_INDX*

*Tens digit of the hours = [(counter / 36000) % 24] / 10 + CHAR\_INDX*

The characters produced from the previous algorithm can be used to display the time to the user. In order to display time, a printing algorithm has to be designed to use the UART\_state data as well as output queue. The algorithm prints one character at a time. The state diagram in Figure 7 shows how the printing algorithm works.



Figure : state diagram for the character printing algorithm.

The algorithm can also be explained in structed English as follows:

Printing algorithm

IF the UART\_state = busy THEN

Enqueue the character to be sent on the output queue

ELSE

Set UART\_state = busy

Put the character to be sent in UART0 data register

ENDIF

In case a string needs to be sent, this algorithm can be called on a loop and repeated to each character on the string until a null is encountered.

### Command algorithms

In this section the routines for the three commands: “TIME”, “SET”, and “GO” will be discussed. When the “TIME” command is requested by the user, the system should first check that there is no second token. Next, it displays the time to the user in the format (hh:mm:ss). Then, it needs to go to the next line then go back to the control module. The “SET” command needs to check that the second token has the format (hh:mm:ss) and also make sure that there is not third token. Then, it extracts the time from the second token and updates the time of day counter. Next, it goes to the next line and goes back to the control module. Finally, the “GO” command should check that there is no second token. Then, it displays the stop watch and changes the machine state to SW. There is no need to go to the next line because the stop watch will be overwriting itself every one-tenth of a second. The control module will be responsible for maintaining the stopwatch until a character is entered. The following are structured English algorithms of the three commands.

TIME command algorithm

*IF the second token of the input string is null THEN*

*Perform the get time algorithm for the time of day*

*Display the time of day*

*ELSE*

*Send ‘?’ to the user*

*ENDIF*

*Go to the next line*

SET command algorithm

*IF the third token is null THEN*

*IF the second token has the format hh:mm:ss THEN*

*Extract the time information from the second token*

*Update the time of day counter*

*ELSE*

*Send ‘?’ to the user and go to the next line*

*ENDIF*

*ELSE*

*Send ‘?’ to the user and go to the next line*

GO algorithm

*IF the second token is null THEN*

*Perform the get time process for the stop watch counter*

*Display the stop watch*

*Machine state = SW*

*ELSE*

*Send ‘?’ to the user and go to the next line*

*ENDIF*

## Messaging system

The initialization algorithms for the SYSTICK module is discussed in great detail in the course website. However, the interrupt service routine needs to be designed. One main issue that needs to be addresses is that there is no fixed period that will produce one-tenth of a second. The machine maximum number of ticks is 2^24 = 16,777,216. So, we can tell the machine to notify us every 1,677,721 ticks and assume that elapsed time is equal to one-tenth of a second. However, one-tenth of a second is actually 1,677,721.6 ticks long. This means that the time in the machine will be drifting away from the actual time.

To resolve time drifting, we note that 2^24 ticks will cause one second to elapse. So, if we make the machine cause an interrupt every 2^14 ticks, then a second will have elapsed after 2^10 ticks. This means that one-tenth of a second will elapse every 102.4 ticks. Using this, we can tell the interrupt service routine to count the following numbers before enqueuing into the input queue that one-tenth of a second has elapsed:

Time adjust list = 102, 102, 103, 102, 103 ticks.

So, after 102 ticks enqueue a SYSTICK, then count to 102 again and enqueue, then counter to 103 and enqueue, and so on until the end of the previous list. Then you can loop back through the list again.

The interrupt service routine algorithm is described below in structured English format keeping in mind the time drifting issue.

SYSTICK Interrupt Service Routine algorithm

Increment the elapsed ticks counter

IF elapsed ticks counter = entry from the time adjust list indexed by the time adjust index THEN

Increment the time adjust index

Reset the elapsed ticks counter

Enqueue a SYSTICK entry into the input queue

ENDIF

## Time server

The initialization algorithms for the SYSTICK module is discussed in great detail in the course website. However, the interrupt service routine needs to be designed. One main issue that needs to be addresses is that there is no fixed period that will produce one-tenth of a second. The machine maximum number of ticks is 2^24 = 16,777,216. So, we can tell the machine to notify us every 1,677,721 ticks and assume that elapsed time is equal to one-tenth of a second. However, one-tenth of a second is actually 1,677,721.6 ticks long. This means that the time in the machine will be drifting away from the actual time.

To resolve time drifting, we note that 2^24 ticks will cause one second to elapse. So, if we make the machine cause an interrupt every 2^14 ticks, then a second will have elapsed after 2^10 ticks. This means that one-tenth of a second will elapse every 102.4 ticks. Using this, we can tell the interrupt service routine to count the following numbers before enqueuing into the input queue that one-tenth of a second has elapsed:

Time adjust list = 102, 102, 103, 102, 103 ticks.

So, after 102 ticks enqueue a SYSTICK, then count to 102 again and enqueue, then counter to 103 and enqueue, and so on until the end of the previous list. Then you can loop back through the list again.

The interrupt service routine algorithm is described below in structured English format keeping in mind the time drifting issue.

SYSTICK Interrupt Service Routine algorithm

Increment the elapsed ticks counter

IF elapsed ticks counter = entry from the time adjust list indexed by the time adjust index THEN

Increment the time adjust index

Reset the elapsed ticks counter

Enqueue a SYSTICK entry into the input queue

ENDIF

## Window manager

The initialization algorithms for the SYSTICK module is discussed in great detail in the course website. However, the interrupt service routine needs to be designed. One main issue that needs to be addresses is that there is no fixed period that will produce one-tenth of a second. The machine maximum number of ticks is 2^24 = 16,777,216. So, we can tell the machine to notify us every 1,677,721 ticks and assume that elapsed time is equal to one-tenth of a second. However, one-tenth of a second is actually 1,677,721.6 ticks long. This means that the time in the machine will be drifting away from the actual time.

To resolve time drifting, we note that 2^24 ticks will cause one second to elapse. So, if we make the machine cause an interrupt every 2^14 ticks, then a second will have elapsed after 2^10 ticks. This means that one-tenth of a second will elapse every 102.4 ticks. Using this, we can tell the interrupt service routine to count the following numbers before enqueuing into the input queue that one-tenth of a second has elapsed:

Time adjust list = 102, 102, 103, 102, 103 ticks.

So, after 102 ticks enqueue a SYSTICK, then count to 102 again and enqueue, then counter to 103 and enqueue, and so on until the end of the previous list. Then you can loop back through the list again.

The interrupt service routine algorithm is described below in structured English format keeping in mind the time drifting issue.

SYSTICK Interrupt Service Routine algorithm

Increment the elapsed ticks counter

IF elapsed ticks counter = entry from the time adjust list indexed by the time adjust index THEN

Increment the time adjust index

Reset the elapsed ticks counter

Enqueue a SYSTICK entry into the input queue

ENDIF

# Data Dictionary

Tod = 0..4,294,967,294 \* this is the time of day counter \*

Stp = 0..4,294,967,294 \* this is the stop watch timer \*

Machine\_state = [SW | NS]

In\_buffer = {character}128

Character = 0..255

BUFFER\_MAX = 128 \* maximum size of the input buffer \*

Commands = “TIME” + “SET” + “GO”

Tokens = {string}3 \* these are the tokens of the input string \*

String = {character}128

MAX\_TOKEN = 3 \* we cannot have 3 or more tokens (max is 2 for set)\*

time = hours + minutes + seconds + tenths-of-second

hours = 0..23

minutes = 0..59

seconds = 0..59

tenths-of-second = 0..9

\* this is the data structure that contains the time to be displayed \*

UART\_state = [ BUSY | IDLE ]

MAX\_WAIT = 2^24 \*maximum possible period for SYSTICK \*

USED\_PERIOD = 2^14 \* the used SYSTICK period \*

T\_adj = “102” + “102” + “103” + “102” + “103”

\* this is the list used to avoid time drifting \*

T\_adj\_cntr = 0..103 \* this is the ticks counter used in SYSTICK module \*

T\_adj\_andx = 0..4 \* used to index through the time adjust list (t\_adj) \*

Queue entry = type + character

Type = [ SYSTICK | UART ]

Queues = Input queue + Output queue

Input queue = head + tail + counter + {queue entries}32

Output queue = head + tail +counter + {queue entries}32