Palouse RoboSub Club
Microcontroller Library Documentation

By Ryan Summers and James Irwin

Table of Contents:

Introduction

The purpose of the microcontroller library project is to simplify the structure of the microcontroller code and eliminate redundancy within the existing codebase in use on the submarine. With a restructuring of code, modifications are made much simpler by only changing a single code location as opposed to modifying multiple functions. This implementation also eliminates a massive header file of #define declarations of registers on the microcontroller. The library will need to be continually maintained and updated as the microcontrollers are updated. The current library is designed for a PIC32MX250F128B microcontroller (12/22/2014).

The purpose of this documentation is to allow future club members to update the microcontroller code base as needed. This document contains in-depth descriptions of the functionality of all processes within the library and is intended specifically for users maintaining and updating the library. This document is NOT intended for end users of the library in microcontroller design implementation. Please refer to the microcontroller design documentation for that purpose.

The main topics this document will cover will be the fundamental data structures that the library depends on, timer functionality, UART channel functionality, I2C channels, SPI, and ADC (analog to digital converter) implementation. The main data structure underlying all implementation is a simple queue, which structure is utilized within the UART, timer, SPI, ADC, and I2C functions. There are two primary 'layers' within the library. These layers are referred to as a low layer and an upper layer. The low layer contains most of the fundamentals of functionality, typically involving data packets as simple bytes. The upper layer implements a conceptual idea of packet management within the communication channels.

The primary structure of the library utilizes the system.h file as the primary file encapsulating all other header files. Each area of functionality is then given its own specific header and source file which draws from the system.h file. The system.h file holds any information that is utilized by multiple other areas. Typically, this file is used for variable declarations.

Fundamentals

Queue

Overview

The queue is a structure utilized within all processes that utilize interrupts and communication between the microcontroller and a peripheral device. The queue utilized within the library is not dynamic, however, as the microcontroller does not support dynamic allocation of memory. To combat this, a circularly linked array has been implemented with a defined size in place of the dynamic queue. A queue can be specified to have as many elements as desired internally. Default implementation within other areas of functionality will typically utilize arrays of size 100, or another size deemed appropriate for the task. A queue object has been created to help understandability within the code.

Queue Objects

The queue object is composed of 5 distinct elements:

```
typedef struct QUEUE {
    uint8 *buffer;
    uint buffer_size;
    uint QueueStart;
    uint QueueEnd;
    uint numStored;
  } Queue;
```

The *buffer element is a pointer to the memory that the queue utilizes. Typically, the address of the array the queue is using will be stored within this variable. The second element specifies the number of elements that the queue has access to, which typically is the size of the array in use. The QueueStart variable represents the index of the first element of the queue. As the queue is a circularly linked array, the start position of the queue will not necessarily be the first location of the array. Consequently, the QueueEnd variable stores the index of the last element in the queue. There is no guarantee that QueueStart index is smaller than QueueEnd, as the structure is cyclical. Any data not contained between QueueStart and QueueEnd indices is not currently in use by the queue, and the data stored in these locations is unknown. The numStored variable represents the amount of elements that are currently stored in the queue. This variable is redundant, as calculations can be down utilizing other variables within the object to calculate the number of stored elements, but this variable allows for simplicity within the code design. The variable type (uint8 and uint) are definitions contained within the global header file, system.h. uint8 is an 8 bit unsigned integer which is being utilized as one byte of data. Uint is simply an unsigned integer.

Queue Functions

Queue Creation (Initialization)

Definition:

Queue create_queue(uint8* buffer, uint buffer_size);

Specification:

This function will create and return a queue object with the supplied parameters. It will automatically initialize all varaibles and prepare the queue for usage.

Input:

Uint8 *buffer: This variable is a pointer to the memory to be used by the queue

Uint buffer_size: this variable is the size of the memory supplied in the *buffer variable. Typically, this is the size of the array in use

Enqueue (push)

Definition:

int enqueue(Queue* queue, uint8* data, uint data_size);

Specification:

This will push a piece of data onto the queue if there is space to hold it. If the queue is full, the data will be completely ignored. The function will copy the data supplied into the queue memory.

Input:

Queue *queue: This variable is the queue object that will be pushed onto.

Uint8 *data: This variable is a pointer to the data that should be pushed onto the queue

Uint Data_size: This variable is the size of the data that is being supplied within the data pointer.

Dequeue (pop)

Definition:

int dequeue(Queue* queue, uint8* output_data, uint data_size);

Specification:

This function will remove a piece of data from the queue with the required specifications. The data is not deleted from the queue memory, as only the queue indices are changed. The data will be handed back with output data pointers, and the function will return an integer for success or failure. Future implementation: return output error codes.

Input:

Queue *queue: this is a pointer to the queue object that is being used.

Uint8 *output_data: this is a pointer to where the data popped off the queue will be stored.

Uint Data_size: This variable contains the amount of data to be popped off the queue.

Low Layer

***Foreword**: In our design of the library, we chose to simplify the management of interrupt service routines (ISR's) through use of function pointers. When a constructor calls for a function_ptr, it is asking for a pointer to a function that the end user will define on their own and supply. This allows for modularity between projects with the library, and allows the ISR's to do anything that the end user would like them to do. In the actual ISR function declarations, the*

additional functionality will only call if there was a supplied function pointer, otherwise it will skip over it and complete the ISR.

Timer

Overview

The timer is the simplest of all the functionalities built into the library. There is only one function acting as a timer constructor. There are a few specifics relating to which timer is being used. On the PIC32 microcontroller, there are type A and type B timers with different functionalities. For our purposes, the only difference is that the type A timers do not have access to the number of clock dividers as the type B timers. Type A timers may only have clock dividers of 1, 8, 64, and 256. Type B timers have access to all range of clock dividers of multiplicity 2. If timer 1 is supplied a clock divider that is not supported by type A, our function will automatically truncate the timer value. This results in a procedurally unknown clock divider. By truncating the supplied timer value by one bit. This is currently untested but believed to give the closest possible acceptable divider. Any other timer is allowed to utilize any clock divider supported.

Objects

Timer objects were deemed unnecessary as timers are representations of basic functions. There are a number of enumerations, however.

```
typedef enum {
    Div_1,
    Div_2,
    Div_4,
    Div_8,
    Div_16,
    Div_32,
    Div_64,
    Div_128,
    Div_256
  } Clock_Divider;
```

This clock divider enumeration provides which scalar should be supplied for the clock dividing parameter of the constructor.

```
  typedef enum {
    Timer_1,
    Timer_2,
    Timer_3,
    Timer_4,
    Timer_5
  } Timer_Type;
```

The timer type enumeration specifies which timer is being used, aka timer 1 (the only type A timer), or timers 2-5.

Functions

Timer Constructor

Definition:

void initialize_TIMER(Clock_Divider divide, uint16 period, Timer_Type timer, void *function_ptr, boolean enable);

Specifications:

The timer constructor should be supplied a clock divider that is appropriate for the timer type being used. If the timer type is timer_1, the clock dividers supported are 1, 8, 64, and 256. Any other values will be truncated to two bits. Any other timer will support a full range of clock dividers.

Input:

Clock_Divider Divide: The supplied value of the clock divider. A clock divider of 2 will divide the default onboard clock by 2.

Uint16 Period: The period that the clock should count to before it fires the interrupt.

Timer_Type Timer: The timer specifically in use, aka Timer_1, or timers 2-5.

Void *function_ptr: The supplied function pointer that will allow for additional functionality within the ISR. If this is supplied NULL, there is a default ISR that will fire for information management. This pointer is solely for additional functionality desired within the ISR. A standalone function should be created on the end-user side to fill this parameter.

Boolean Enable: This variable will enable or disable the timer at creation.

UART

Overview

The UART functionality depends on the fundamental queue structure specified earlier. Within the UART, there is a queue for received information (rx queue), and a queue for transmission (tx queue). Both of these queues are crucial for ISR functionality. The way that the ISRs work is defined into two separate processes. There is an active process that occurs when the ISR is fired, and there is a background process for information management. The purpose of two processes is to keep the overhead time involved within the ISR as small as possible, as interrupts are disabled within all ISRs in the library. If another ISR fires while one is being handled, it will wait until interrupts are re-enabled to be handled. In the case where there is time-sensitive information involved, overhead needs to be minimized. The UART ISR will only pop information from communication registries on the microcontroller and push them onto the received queue of the UART object, or it will grab data from

the transmit queue and place it into the transmission registry on the microcontroller. The background process is where actual data processing occurs. In the background process, data within the received and transmission queues will be modified and prepared. The library functions do not do anything with data in the received and transmit queues, but provide function pointers to allow for end-user implementation.

Objects

```
typedef struct UART_DATA {
    Queue Rx_queue;
    Queue Tx_queue;
    boolean Tx_is_idle;
} Uart_Data;
```

The Uart_Data object contains two queue objects and a Boolean flag. The queue objects are for transmission and receiving of information within the ISR. The Boolean flag indicates whether or not the Tx queue has information within it.

```
typedef enum {
    UART1,
    UART2
} Uart;
```

The Uart enumeration specifies which UART channel on the microcontroller is being used. There are currently only two channels available on the PIC32MX250F128B.

Functions

Uart Constructor

Definition:

Uart_Data* initialize_UART(uint speed, uint pb_clk, Uart which_uart, uint8 *rx_buffer_ptr, uint8 rx_buffer_size, uint8 *tx_buffer_ptr, uint8 tx_buffer_size, boolean tx_en, boolean rx_en, void* rx_callback, void* tx_callback);

Specification:

The constructor function creates a UART communication channel with the supplied variables, allowing for data management as specified within the tx and rx function pointers. Agreed upon channel speeds between the computer and the microcontroller must be supplied within the speed and pb_clk variables to synchronize data transmissions. This function will hand back a pointer to the UART_Data object constructed. The data must remain internal as a global variable within the library function for use by the ISRs, send, and transmit functions.

Input:

uint speed: The clock speed used to calculate baud rate.

uint pb_clk: Clock speed utilized in baud calculation

Uart which_uart: This variable specifies which UART channel is to be used.

uint8 *rx_buffer_ptr: This is a pointer to memory allocated for use by the rx queue.

uint8 rx_buffer_size: This specifies the size of memory pointed to for the rx queue.

uint8 *tx_buffer_ptr: This is a pointer to the memory allocated for use by the tx queue.

uint8 tx_buffer_size: This specifies the size of the memory pointed to for the tx queue.

boolean tx_en: This enables or disables transmission.

boolean rx_en: This enables or disables receiving on the channel.

void* rx_callback: A function pointer for data handling of information within the received queue.

void* tx_callback: A function pointer for data handling of information to be placed within the tramsission queue.

Transmit

Definition:

int send_UART(Uart channel, uint8 data_size, uint8 *data_ptr);

Specification:

This function will send a piece of information on the specified uart channel. Memory is copied into the queue, so data specified within the data_ptr can be changed any time after the function call. The function will return a 0 upon success, and a 1 upon failure. This function will place data onto the tx queue, while a separate process in the ISR will send the data in the queue.

Input:

Uart channel: This specifies which UART should be used, aka UART1, or UART2.

uint8 data_size: This specifies the size of data pointed to by the data_ptr variable

uint8 *data_ptr: A pointer to the data to be copied into the tx queue.

Receive

Definition:

int receive_UART(Uart channel, uint8 data_size, uint8 *data_ptr);

Specification:

This function will remove information from the received queue and place it into the specified pointers in the function call. This is essentially a pop function for the rx queue.

Input:

Uart channel: This specifies which UART should be used, aka UART1, or UART2.

uint8 data_size: This specifies the size of data pointed to by the data_ptr variable

uint8 *data_ptr: A pointer to the data to be copied into the rx queue.

I2C

Overview

The I2C is a communication channel utilized for communication channels between external sensors and the microcontroller. **[INSERT BACKGROUND INFORMATION ABOUT I2C]**

Objects

```
typedef struct I2C_NODE {
    uint8 device_id; //a unique identifier for the device
    uint8 device_address; //I2C address for device
    uint8 sub_address; //internal device address
    uint8* data_buffer; //data buffer to store the received data or write data
    uint8 data_size; //how much data to send/read from device (must be <= buffer size)
    I2C_MODE mode; //reading or writing?
    void (*callback) (struct I2C_NODE); //callback function
  } I2C_Node;
```

[INSERT INFORMATION]

```
typedef struct I2C_DATA {
    Queue Rx_queue;
    Queue Tx_queue;
    boolean Tx_is_idle;
  } I2C_Data;
```

[INSERT INFORMATION]

```
typedef enum {
    STARTED,
    DEV_ADDR_W_SENT,
    SUB_ADDR_SENT,
    DATA_SENT,
    RESTARTED,
    STOPPED_TEMP,
    STARTED_TEMP,
    DEV_ADDR_R_SENT,
    DATA_RECEIVED,
    ACK_SENT,
    NOACK_SENT,
    STOPPED
```

```
} I2C_STATE;
```
[INSERT INFORMATION]

```
typedef enum {
    I2C1,
    I2C2
} I2C_Channel;
```
[INSERT INFORMATION]

Functions

I2C Constructor

Definition:

Specification:

[INSERT INFORMATION]

Input:

[INSERT INFORMATION]

I2C Send

Definition:

Specification:

[INSERT INFORMATION]

Input:

[INSERT INFORMATION]

I2C Background Process

Definition:

Specification:

[INSERT INFORMATION]

Input:

[INSERT INFORMATION]

ADC

Overview

Objects

Functions

Upper Layer

Packet Communication

Overview

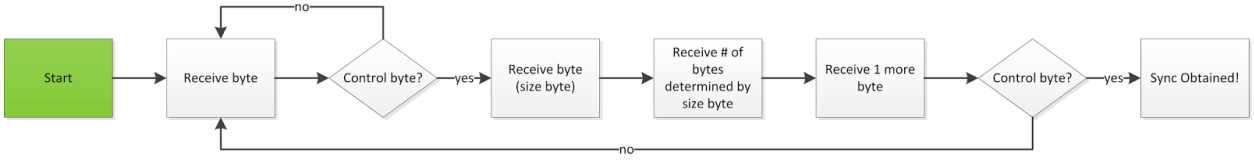
The packet-based communication protocol is primarily used for communication between a microcontroller and computer, but it could also work for inter-microcontroller communication. It is using the UART protocol as the underlying data transport mechanism, but this library could be expanded to use other protocols such as SPI for faster communication if desired.

The protocol sends data as a series of bytes, logically grouping them together into packets. The figure below shows the packet format.

The first byte in a packet is known as the *Control Byte*, which signifies the start of the packet and is used for synchronization. It is the newline character *'\n'*, or *0x0A*. The next byte is the *Size Byte*, describing how large the remainder of the packet is. This byte must be a size between (0,1)-255. The next byte(s) contain the data. There is not a hard specification for how the data must be organized, but it is recommended that the first few bytes be description bytes that describe the rest of the data, and the remainder are data bytes.

Packet Synchronization

Packet synchronization is necessary on the receiving end to ensure packets are interpreted properly. Under normal operation, sync should be obtained during startup of the communication channel, and should remain in sync. However, sync can be regained if packets data is scrambled or a program crashes, but any data sent in the meantime will be lost. The basic algorithm for obtaining sync is described in the figure below.



Bytes are received and thrown away until the control byte is found. Once found, the next byte received is interpreted as the size byte. Then a number of bytes are received, as determined by the size byte. One more byte is then received, which should be the control byte if we are properly in sync.

If the byte is not the control byte, we keep looking again for the control byte. If it is, then sync has been obtained.

After sync has been obtained, when packets are received the first byte is checked to see if it is the control byte. If not, the synchronization algorithm is performed again. This ensures that the communication channel remains in synchronization.

Objects

Functions

Sensors

Overview

This component is an abstraction for interfacing with sensors. It allows for the easy construction of individual sensor functions. It supports any underlying data transport protocol, including I2C and ADC-based sensors. How do we do this? I've have no clue. But it would be really cool yeah?

Objects

Functions