College of Engineering
UNIVERSITY OF GEORGIA

ECSE 4235 Library

Table of Contents

Library Overview

This is a C library for the Raspberry Pi 4 (RP4) written by the University of Georgia (UGA) Embedded Systems II (ECSE 4235) course. The intention of the project is to provide an open-source framework that future students can build on to expand the functions of the library. The source code for the library is written using the ARM Assembly language.

The library is organized into several modules that provide access to GPIO and other IO functions on the Broadcom BCM 2711 chip. The *system module* is used to provide access to provide information about the RP4 or control system wide IO. The *delay module* contains functions that allow the user to delay the execution of a program for a specified amount of time. The *GPIO module* allows the user to control the GPIO interface. Each GPIO can be set to a specified function, set the output state, or read the input state.

It is important to note that any programs that access peripheral devices accessed using */dev/mem*, such as Pulse Width Modulation (PWM), Serial Peripheral Interface (SPI) communications, or Inter-Integrated Circuit (I2C) protocols, must be ran with root permissions. It is also important to note that any functions from the GPIO module use GPIO number and *NOT* the board pin number.

Revision History

Date	Authors	Update	Version
[04/29/2024]	Olivia Beattie Spring 2024 Class	Created document. Added overview, installation instructions, and documentation for system module, delay module, and GPIO module.	1.0.0

Library Setup and Use

Directory Structure

The Library repository is organized into directories to facilitate development and testing; the repository is composed of five directories that contain the code for each component (*bin, docs, obj, src, and test*) and the remaining important files in the top level.

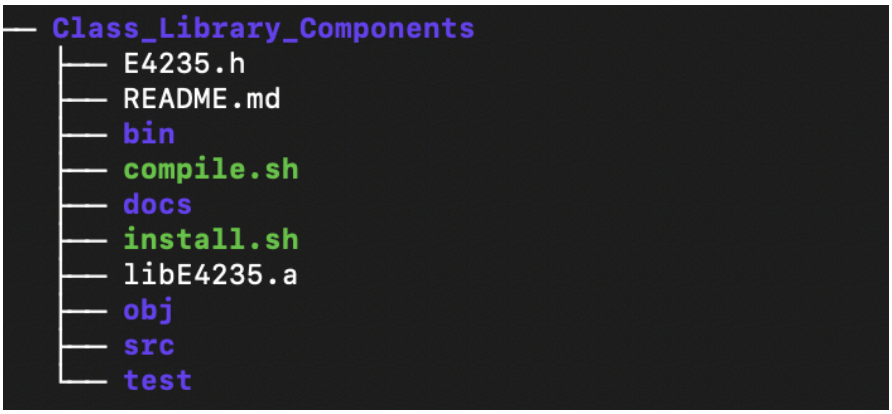

Figure 1. Repository Structure

The *src* directory contains all of the source code, where each individual function has its own assembly file. The *bin* and *obj* directories contain the compiled executables and objects that are created when compiling and installing the library. The *test* directory stores programs written in both C and assembly that validate the functionality of each library component. The *docs* directory holds the original documentation created by the class members for each library component.

The header file, *E4235.h*, consists of all the required define and include statements for the library; each function's signature is contained in the header using the *extern* keyword. The library file, *libE4235.a*, is the final static library that is created from compiling the source code.

Compilation and Installation

The library can be obtained from GitHub using the command below. The repository does *not* need to be cloned into the current project directory, but can be placed in any preferred directory on the system.

```
git clone git@github.com:Herring-UGAECSE-4235/Class_Library_Components.git
```

The library consists of a single header file and a source file for each function. Once the library has been obtained from the GitHub, it can be installed to the system using the provided *compile* and *install* scripts. Inside of the repository, run the following commands.

```
cd [PATH]/Class_Library_Components
./compile.sh
sudo ./install.sh
```

The *compile* script compiles each individual program in the src folder and stores its object file in the *obj* directory. Then, the libE4235.a library is generated by linking every object file in the *obj* directory together. The *install* script then copies the library file into /usr/local/lib on the system and copies the header file into /usr/local/include. This allows gcc to find the header and library files without explicitly specifying the relative path.

Using the Library

Once the library has been installed, it is ready to be used in both C and assembly programs. In order to call the library from a C program, the header file, *E4235.h*, must be included and the library flag *-lE4235* needs to be added when compiling with *gcc*. In order to use the functions in an assembly program*, only the *-lE4235* flag must be included in the compilation command.

```
gcc file.c -o file -lE4235
```

*It is important to note that when calling the functions in assembly, the first 4 arguments must be stored in registers *R0* - *R3*, and any remaining arguments must be stored on the stack. All functions will place the return value into register *R0*.

Updating the Library

If any updates are made to the library, the user should change into the directory where the library was originally installed, use git to pull any changes, and re-run the *compile* and *install* scripts.

```
cd [PATH]/Class_Library_Components
git pull
./compile.sh
sudo ./install.sh
```

System Module

The system module provides functions that allow the user to get information regarding the CPU and set system wide modes or parameters.

Functions

int **E4235_KYBdeblock** (int state)

int **E4235_WhatAmI** ()

Function Documentation

int E4235_KYBdeblock (int state)

This function controls the blocking status of the standard input file. Setting this function to ON results in standard input reads returning immediately. Setting this function to OFF results in a standard input read blocking the program from continuing until an input is read.

Parameters

state - the state of the standard input deblocking

Returns

error_code - an integer that describes if the function executed successfully or not
0 - deblocking successfully set
1 - state integer out of bounds error

int E4235_WhatAmI ()

This function is used to get information about the central processing unit for the Raspberry Pi 4. Upon the call of the function, the program will return the maximum frequency of the CPU in Hertz.

Returns

frequency - the frequency of the processor in Hz

Examples

`E4235_KYBdeblock(1)`
Unblocks standard input. Any scanf or read calls will return immediately.

`E4235_KYBdeblock(0)`
Blocks standard input. Any scanf or read calls will wait to return until a newline character or carriage return is read.

Delay Module

The delay module contains functions that allow the user to delay the execution of the program for a specified amount of time.

Functions

void **E4235_DelayNano** (int count)

void **E4235_DelayMicro** (int count)

void **E4235_DelayCenti** (int count)

void **E4235_DelayMilli** (int count)

void **E4235_DelaySec** (int count)

void **E4235_DelayMin** (int count)

Function Documentation

```
void E4235_DelayNano ( int count )
```

To delay the continuation of a program by the specified number of nanoseconds.

Parameters

count - the number of nanoseconds to delay for

```
void E4235_DelayMicro ( int count )
```

To delay the continuation of a program by the specified number of microseconds.

Parameters

count - the number of microseconds to delay for

```
void E4235_DelayCenti ( int count )
```

To delay the continuation of a program by the specified number of centiseconds.

Parameters

count - the number of centiseconds to delay for

void E4235_DelayMilli (int count)
To delay the continuation of a program by the specified number of milliseconds.
Parameters **count** - the number of milliseconds to delay for

void E4235_DelaySec (int count)
To delay the continuation of a program by the specified number of seconds.
Parameters **count** - the number of seconds to delay for

void E4235_DelayMin (int count)
To delay the continuation of a program by the specified number of minutes.
Parameters **count** - the number of minutes to delay for

Examples

```
E4235_Delaysec(30)
```

Delays the execution of the program for 30 seconds.

GPIO Module

The GPIO module allows the user to control the GPIO interface. Each GPIO can be set to a specified function, set the output state, or read the input state. A PWM signal can be set on certain GPIOs specified by the BCM2711 document [1].

Functions

int **E2435_Select** (int GPIO, int state)

int **E4235_Read** (int GPIO)

int **E4235_Write** (int GPIO, int state)

int **E4235_PWM_SET** (int GPIO, int FREQ, int DUTY)

int **E4235_PWM_Enable** (int GPIO, int enable)

int **E4235_multiread** (int [] GPIO, int length)

void **E4235_multiwrite** (int [] GPIO, int length, int state)

Function Documentation

`int E4235_Select ( int GPIO, int state )`

Select is used to set the state of the specified GPIO to either input or output.

Parameters

GPIO - the GPIO pin to set. Must be valid from 0-29. Note that this is NOT the pin number

Returns

error_code - an integer that describes if the function executed successfully or not
-1 - error occurred and will print an error message

`int E4235_Read ( int GPIO )`

Read is used to read the current value of a specified GPIO pin. Note that only valid values of the GPIO number can be read, else an error message is outputted.

Parameters

GPIO - the GPIO pin to read from. Must be valid from 0-29. Note that this is NOT the pin number

Returns

state - the state of the GPIO
0 - a LOW detected at the pin
1 - a HIGH detected at the pin
error_code - an integer that describes if the function executed successfully or not
-1 - error occurred and will print an error message

int E4235_Write (int GPIO, int state)

Write is used to write a HIGH/LOW value to a given GPIO pin. It will stay HIGH/LOW unless overridden by another write invocation or if the pi is turned off.

Parameters

state - the state of the standard input deblocking

Returns

error_code - an integer that describes if the function executed successfully or not
0 - deblocking successfully set
1 - state integer out of bounds error

int E4235_PWM_SET (int GPIO, int FREQ, int DUTY)

This function sets the parameters of a hardware PWM signal on one of the Pi's four PWM pins: GPIOs 12, 13, 18, and 19. The user can choose a frequency and duty cycle to run on any of the four PWM pins. The frequency input can be anything greater than 5 Hz; however, the square wave will typically break down past 10MHz. The duty cycle can be any integer between 0 and 100 and represents what percent of the cycle is high. A duty cycle of 0 represents a continuously low output while a duty cycle of 100 represents a continuously high output. The user would use this function to generate a square wave or drive a load at a lower voltage. For example, the user might wish to drive a motor at a variable speed; choosing different duty cycles would change the speed of the motor.

Parameters

GPIO - the GPIO to run the PWM on
FREQ - the frequency of the signal
DUTY - the duty cycle of the signal

Returns

error_code - an integer that describes if the function executed successfully or not
0 - successful set of PWM signal
1 - GPIO is an invalid pin
2 - FREQ is out of range
3 - DUTY is out of range

`void E4235_PWM_Enable (int GPIO, int enable)`

This function enables/disables a PWM output on a given GPIO pin. The GPIO can be one of the four PWM pins: 12, 13, 18, or 19. Enable is an integer that determines the output state of the GPIO pin. 0 disables the PWM output while any positive value enables the PWM output (typically 1 is used to enable the output). The user would use this function to turn a PWM signal on or off for a given GPIO. All PWMs are initialized off on Pi bootup.

A note on the BCM2711 PWM implementation: The Pi PWM pins are run by two channels: channel 0 runs pins 12 and 18 while channel 1 runs pins 13 and 19. Therefore, if the user enables/disables pin 12, it will also enable/disable pin 18 since the code is actually enabling/disabling PWM channel 0. In all, there can only be two separate PWM outputs at any given time: one on pins 12 and 18 with channel 0 and one on pins 13 and 19 with channel 1.

Parameters

GPIO - the pin that is being enabled or disabled
enable - enables or disables the PWM output

Returns

error_code - an integer that describes if the function executed successfully or not
0 - successful enable/disable of a GPIO
1 - GPIO is an invalid pin
2 - enable is out of range

`uint16_t E4235_multiread ( int [] GPIO, int length )`

The multiread function is designed to read multiple GPIO values at the same time, mimicking a bus.

Parameters

GPIO - array of pin numbers
- If a series of numerically ordered pins are wanted, the format [#, '-', #] can be used inside the array
- Cannot exceed 16 pin

length - the length of the array

Returns

state - the state of the GPIO
0 - a LOW detected at the pin
1 - a HIGH detected at the pin
error_code - an integer that describes if the function executed successfully or not
-1 - error occurred and will print an error message

```
void E4235_multiwrite ( int [] GPIO, int length, uint16_t state )
```

The multiwrite function is used to be able to write to multiple GPIO pins at the same time, mimicking a bus.

Parameters

GPIO - array of pin numbers
- If a series of numerically ordered pins are wanted, the format [#, '-', #] can be used inside the array
- Cannot exceed 16 pin

length - the length of the array

state - the state to write to the pins (HIGH or LOW)

Examples

`E4235_Select(5, 1)`
Set GPIO 5 function to OUTPUT.

`E4235_Write(5, 1)`
Set GPIO 5 to HIGH.

`E4235_Read(5)`
Read the input from GPIO 5.

`E4235_PWM_SET(12, 1000, 50)`
Sets GPIO 12 to output a PWM signal with 1000 Hz frequency and a duty cycle of 50% on and 50% off.

`E4235_PWM_EN (12, 0)`
Disables pin 12.

`E4235_PWM_EN (13, 1)`
Enables pin 13.

`E4235_PWM_EN (12, 1); E4235_PWM_EN (18, 0)`
Disables the output on pins 12 and 18 (the outputs of 12 and 18 are tied together).

Example Programs

WhatAmI

A program called *WhatAmI.c* that returns the maximum frequency of the CPU.

```c
#include <stdio.h>
#include "E4235.h"

int main() {
   int freq = 0;
   freq = E4235_WhatAmI(); // gets frequency using WhatAmI
   printf("freq = %d\n", freq); // prints frequency to terminal
} // main
```

BlinkLED

A program called *blink.c* that turns an LED *on* and *off* at a specified delay time.

```c
#include "E4235.h"
#include <stdio.h>

int main() {
   E4235_Select(12, 1); // set GPIO 12 to output

   while (1) {
       E4235_Write(12, 1);
       E4235_Delaynano(500000);
       E4235_Write(12, 0);
       E4235_Delaynano(500000);
   }
}
```

PWM

A program called *PWM.c* that sets a 10 kHz, 50% duty cycle PWM on GPIO 12.

```c
#include <stdio.h>
#include <stdlib.h>
#include "E4235.h"

int main() {
   E4235_PWM_Set(12, 100000, 50);
   E4235_PWM_Enable(12, 1);
} // main
```

References

[1] BCM2711 Raspberry Pi Documentation
https://www.raspberrypi.com/documentation/computers/processors.html#bcm2711

[2] BCM2835 Library Reference
https://www.airspayce.com/mikem/bcm2835/index.html