LAB EXERCISE 1 2023 CAR PARK CATALOGS

A. Car Park Main Code

Car Park Main Code

```
#include "mcc_generated_files/mcc.h"

#define ENTRANCE PORTAbits.RA2
#define EXIT PORTAbits.RA1  // ENTRANCE and EXIT are defined to represent the states
                                of entrance and exit signals connected to pins RA2
                                and RA1, respectively.

#define CAR_FULL 20  // CAR_FULL represents the maximum number of cars allowed.

#define CAR_FULL_LIGHT LATBbits.LB5 // CAR_FULL_LIGHT is used to control an
                                        indicator, typically an LED, which is used to
                                        indicate when the car park is full.

#define decrease_cars() do { if (num != 0) num--; if(num <= CAR_FULL)
CAR_FULL_LIGHT=0;} while(0)

void Initial(void);  //This function is used for the initialization of the microcontroller. It
                        configures port settings, sets initial values, and waits for a brief period.

void delay_1s(void);  // This function creates a delay of approximately 1 second using a
                         loop that calls the __delay_ms function (delay in milliseconds)
                         multiple times.

void increase_cars(void);  //This function is used to increment the car count (num). If the
                              car count exceeds the predefined limit (CAR_FULL), it sets
                              CAR_FULL_LIGHT to 1 to indicate that the parking is full.


unsigned char num = 0;

void main(void)
{
  // Initialize the device

  SYSTEM_Initialize();
  Initial();
```

```
  while(1)
  {
    while(!ENTRANCE && !EXIT);

    if(ENTRANCE) // If ENTRANCE is active (indicating a car entering), it calls
                          increase_cars() to increment the car count.
    {

      increase_cars();

    }
    else
    {
      decrease_cars();
    }


    while(ENTRANCE||EXIT);  //After processing entrance or exit, it waits until both
                                    ENTRANCE and EXIT signals are inactive.


    LATB= LATB & 0x20;    //1000 0000  -> 0010 0000  display
    LATB= LATB | num;  // count


  }
}




void Initial(void)
{

  TRISB = 0x00;

  LATB= 0x00;
  delay_1s();
  delay_1s();
  LATB= 0xFF;
  delay_1s();
  delay_1s();
  LATB= 0x00;

}
void delay_1s(void)
{
  unsigned char i;
```

```
   for(i=0;i<25;i++)
            __delay_ms(40);
}


void increase_cars(void)
{

   if(num != 31)
      num++;
   if(num > CAR_FULL)
      CAR_FULL_LIGHT=1;

}
```

B. The difference between an ‘IF’ and a ‘WHILE’

Explain the difference between checking/testing a switch value using an ‘IF’ statement and waiting(polling) on a specific switch value using a ‘WHILE’ statement. Use code and an English description to illustrate your answer

```
#define SWITCH_PIN PORTBbits.RB0


void main() {
   TRISB0 = 0x FF; // Configure PORTB as an input (assuming it's connected to the switch)


   while (1) {
      if (SWITCH_PIN == 1) {
         // Switch is ON


      }
            else {
         // Switch is OFF


      }
   }
}
```

In this code, we use an 'IF' statement to check the state of the switch connected to the RB0 pin. If the switch is ON (the value of SWITCH_PIN is 1). If the switch is OFF (the value of SWITCH_PIN is 0), it performs a different action. However, this code continuously checks the switch state in a loop.

```
#define SWITCH_PIN PORTBbits.RA0


void main() {
   TRISB0 = 0x FF;  // Configure PORTB as an input (assuming it's connected to the switch)
```

```
  while (1) {
    while (SWITCH_PIN == 0) {
      // Wait for the switch to turn ON
    }

    // Switch is now ON


    while (SWITCH_PIN == 1) {
      // Wait for the switch to turn OFF
    }

    // Switch is now OFF
      }
}
```

In this code, we use a 'WHILE' statement to actively wait(poll) the switch's state. The program waits for the switch to turn ON, waits for the switch to turn OFF. It continuously checks the switch state within the 'WHILE' loops.

Different:

while is a loop statement, it can be looped multiple times. If is a conditional statement and can only be executed in a single pass. A while loop in Programming repeatedly executes a target statement as long as given condition is true.

C. Pre-processor

Explain the function of the pre-processor. Give some examples.

The pre-processor is a macro pre-processor (allows you to define macros) that transforms your program before it is compiled. These transformations can be the inclusion of header files, macro expansions, etc.

All pre-processing directives begin with a # symbol.

For example:

```
#include "mcc_generated_files/mcc.h" //  Use your own program written in
                                        "mcc_generated_files/mcc.h" file

#define ENTRANCE PORTAbits.RA2  //  When the PORTAbits.RA2 appears in the program,
                                   it can be replaced by ENTRANCE. If you want to
                                   modify PORTAbits.RA2, you only need to modify
                                   PORTAbits.RA2 in the macro definition, and you
                                   can achieve the purpose of modifying
                                   PORTAbits.RA2 in the whole file.

#define EXIT PORTAbits.RA1 // When the PORTAbits.RA1 appears in the program, it can
                              be replaced by EXIT. If you want to modify
                              PORTAbits.RA1, you only need to modify PORTAbits.RA1
                              in the macro definition, and you can achieve the purpose
                              of modifying PORTAbits.RA1 in the whole file.


#define CAR_FULL 20 // When the 20 appears in the program, it can be replaced by
                       CAR_FULL. If you want to modify 20, you only need to modify 20
                       in the macro definition, and you can achieve the purpose of
                       modifying 20 in the whole file.

#define CAR_FULL_LIGHT LATBbits.LB5 // When the LATBbits.LB5 appears in the
                                       program, it can be replaced by
                                       CAR_FULL_LIGHT. If you want to modify
                                       LATBbits.LB5, you only need to modify
                                       LATBbits.LB5 in the macro definition, and you
                                       can achieve the purpose of modifying
                                       LATBbits.LB5 in the whole file.
```

D. Function Prototype

Explain the purpose of a function prototype

The Function prototype is necessary to serve the following purposes:

1. Function prototype tells the return type of the data that the function will return.

2. Function prototype tells the number of arguments passed to the function.

3. Function prototype tells the data types of each of the passed arguments.

4. Also, the function prototype tells the order in which the arguments are passed to the function.

Therefore essentially, the function prototype specifies the input/output interface to the function i.e. what to give to the function and what to expect from the function.

Note: The prototype of a function is also called the signature of the function.

Example:

```
void Initial(void) ; // initialization


void Initial(void)
{

   TRISB = 0x00; // Set PORTB as output PORT

   LATB= 0x00; // Set PORTB low initially (All LEDs off)

   delay_1s(); // Delay of 1 millisecond

   delay_1s();
   LATB= 0xFF;  // Set PORTB high initially (All LEDs on)
   delay_1s();
   delay_1s();
   LATB= 0x00;  // Set PORTB low initially (All LEDs off)

}

void delay_1s(void); // Delay

void delay_1s(void)
{
   unsigned char i;
```

```
   for(i=0;i<25;i++) // for i=0 to i<25,i+1
          __delay_ms(40);
}
void increase_cars(void); // the number of cars are increasing

void increase_cars(void)
{

   if(num != 31) // if the number of cars != 1111 1000
      num++; // num + 1
   if(num > CAR_FULL)  // num > 20
      CAR_FULL_LIGHT=1;

}
```

E. Header File

Explain the purpose of a header file.

The header files shipped with the compiler are specific to that compiler version. Header file helps to reduce the complexity and number of lines of code. It also gives you the benefit of reusing the functions that are declared in header files to different.

delay. h consists of functions that generate delays in program execution.

Pin_manager.h displays the possible configurations of the peripheral. Pin Manager Area: The I/O pins of the device can be configured in pin manager in Table view. It also displays the pinout of the device and their functions as a Package view.

Device_config.h is CONFIG registers

Mcc.h oscillator initialize

F. Linker

Research and explain the role of the linker.

The linker uses classes to represent memory ranges in which psects can be linked. Classes are defined by linker options. The compiler driver passes a default set of such options to the linker, based on the selected target device.

```
#include "mcc_generated_files/mcc.h"
#define FLASH_LED PORTAbits.RA0
#define FLASH_FULL_LED LATBbits.LB7

// Function prototype

void Initial(void);

void delay_1s(void);
void Flash_led(unsigned char num);

  void main(void)
{

   SYSTEM_Initialize();
   Initial();

   Flash_led(3); // Function call

}

// Function to flash the LED a specified number of times

 void Flash_led(unsigned char num)
    unsigned char j;

    for(j=0; j<num; j++)
    LATB= 0xFF;
    delay_1s();
    delay_1s();
   LATB= 0x00;
}

void Initial(void)
{
```

```
    TRISB = 0x00;

    LATB= 0x00;
    delay_1s();
    delay_1s();
    LATB= 0xFF;
    delay_1s();
    delay_1s();
    LATB= 0x00;

}
void delay_1s(void)
{
    unsigned char i;

    for(i=0;i<25;i++)
            __delay_ms(40);
}
```

G. Between a normal function and a function-like macro

Explain the difference between a normal function and a function-like macro.

1. Compilation Time vs. Runtime:

Normal Function: A normal function is compiled just once, and its code is generated by the compiler. It is executed at runtime when called.

Function-Like Macro: A function-like macro is a piece of code that is substituted directly into your code by the preprocessor before compilation. There's no runtime function call overhead, but it can lead to larger compiled code because the code is duplicated wherever the macro is used.

2. Arguments Handling:

Normal Function: Functions accept arguments and can have a variable number of arguments if you use variadic functions (e.g., printf).

Function-Like Macro: Macros can also accept arguments, and these arguments are replaced directly in the macro's code. Macros do not check the types or validity of their arguments, which can lead to unexpected behavior if not used carefully.

3.Type Safety:

Normal Function: Functions are type-safe. The compiler enforces type checking, and you get compile-time errors if there's a type mismatch.

Function-Like Macro: Macros are not type-safe. They don't perform type checking, which can lead to errors that are harder to catch because they only appear during compilation or even at runtime.

4.Debugging and Error Messages:

Normal Function: Functions provide better debugging information and error messages since the compiler and debugger can relate issues to the function name.

Function-Like Macro: Macros can make debugging more challenging because the code is expanded in place, and error messages might refer to macro expansions, making it less clear where the issue originates.

5. Scope:

Normal Function: Functions have their own scope. Variables declared within a function are typically local to that function.

Function-Like Macro: Macros operate within the scope of where they are used. They don't create new variable scopes.

6. Overhead:

Normal Function: Function calls typically introduce a small amount of overhead due to parameter passing, stack management, and the return mechanism.

Function-Like Macro: Macros are expanded directly into the code, avoiding the overhead of function calls.