Creating a Custom Pause Function

Jeffrey La Favre

October 23, 2015

You have learned in a previous lesson that there is no pause function available in the standard input output library for the **Dev-C++** IDE. When writing code for a robot that operates with a microcontroller, there is a need to have a pause function. Using the C programming language it is possible to create **custom functions**, also known as **subroutines**. In this lesson you will learn about the method used to create a custom function. Specifically, you will study how a pause function can be created using Dev-C++.

- Start **Dev-C++**.
- Open the **File** menu and select **New**. Then select **Source File**.
- Click the mouse in the text window of **Dev-C++** and use the keyboard to enter the following text: **#include <stdio.h>**
- On the next line enter this text: **#include <time.h>** We will need this extra library because it contains functions and other items related to tracking time, which are needed for the custom pause function.
- Open the **File** menu and select **Save**, which opens a **Save As** dialog box.
- In the dialog box, open the drop-down labeled **Save as type** and select **c source files(*.c)**. In the **file name** slot enter this name for the file: **custom pause.** At the top of the dialog box there is a **Save in** slot, which determines where the file will be saved. Make sure you know the location where you are saving your file so that you can find it later. Now click the **Save** button to save your program file.
- Copy the text in the box on the next page and paste it into the text window of **Dev-C++** under the two lines of text you have already entered. Alternatively, you can type the code in yourself, not including text starting with // of each line.
- Click the **Save** button to save the code you just pasted or entered with the keyboard.
- Run the program by opening the **Execute** menu and selecting **Compile and Run**. If there are no errors in the program, a new program window will open (terminal window). You should notice that the program pauses the number of milliseconds entered by you in the terminal window.

```
#include <stdio.h>
#include <time.h>
                                                    // this code provides a means for doing a pause - code by Jeff La Favre

void pause(unsigned int ms);                        //a custom function by the name of pause() will be used in this program
int pause_time;                                     //create a variable with the name of pause_time of the integer type

int main()
{
        while (1)                                   //indefinite loop allows user to run program unlimited number of times
        {
        printf("Enter the number of milliseconds to pause program\n");          //display this on the terminal screen
        printf("Then press the Enter key\n");                                   //display this on the terminal screen
        scanf("%d", &pause_time);                   //get the value entered on keyboard and store it in the variable named pause_time
        printf("Program will pause %d milliseconds\n\n", pause_time);           //display number of milliseconds to pause
        pause(pause_time);                          //pause program for number of milliseconds stored in variable named pause_time
        printf("The pause has finished!\n\n");      //display this on the terminal screen
        }                                           //now return to top of the while loop
}                                                   //end of main() code block

                                                    // now comes the code that creates the custom pause() function
void pause(unsigned int ms)                         //create a custom function named pause() with an argument containing a variable named ms
{                                                   //start of code that runs when the pause function is called
  clock_t ticks1, ticks2, target;                   //create three variables of the data type clock_t
  ticks1 = clock(), ticks2 = ticks1;                //clock() gets the number of system clock ticks since start of execution of program
        target = ms * CLOCKS_PER_SEC / 1000; // CLOCKS_PER_SEC is a constant that stores number of system clock ticks per second
   while ( ( ticks2 - ticks1) < target)
     ticks2 = clock();      //as time goes on ticks2 becomes larger and when ticks2 - ticks1 is equal to target, then program exits this while() loop
}                           //  end of code that runs when pause function is called
```

Let's examine the code so that we can understand how to create a custom function. Furthermore, we should be interested in how this custom function works. The first thing to notice is that two libraries are used: #include <stdio.h> and #include <time.h>. You already know about stdio.h, but time.h is new. This new library has functions for tracking time, which is exactly what we need for creating a pause function.

Then we have this line near the top of the code:

void pause(unsigned int ms);

When we use a custom function, we must list it before the line containing **main()**. The above code provides the notification that a custom function named **pause()** will be used. The function will not return a value when it executes, because **void** is given before the function name. In parentheses we see that this function will take one argument which will be stored in a variable named **ms** of the unsigned integer data type. Unsigned integers can only have positive values.

The next line of code is:

int pause_time;

This creates a variable named **pause_time** of the integer data type. This variable will be used to store the number of milliseconds to pause the program.

Now notice that we have a while(1) loop, which means the program will run indefinitely until it is closed.

Then there are two **printf** functions which display some instructions on the terminal screen. The user is instructed to enter the number of milliseconds to pause the program.

Next we have this line:

scanf("%d", &pause_time);

The scanf function retrieves the value input from the keyboard and in this case stores it in the variable named pause_time. The next line is a printf function that will display the number of milliseconds the user input from the keyboard.

The next line is our custom print function:

pause(pause_time);

Now the computer will pause the number of milliseconds stored in the variable pause_time. This function is not part of the standard libraries included with Dev-C++. So how does pause work? For the answer we need to look at the bottom lines of code, below the end of the code block for the main() function. Here we find the following line:

void pause(unsigned int ms)

This line is nearly identical to the line near the top of the code, but does not contain the semicolon (;) after the parentheses of the function. When this function is used in the main() code block, we need to put a number as an argument between the parentheses. In our code that number is the number stored in the variable named pause_time. Now that same number will be stored in the variable named **ms**, which is the variable named in the definition of pause, like this: (unsigned int ms).

Then there is a code block of five lines, which describe exactly what this function does. The first line is:

clock_t ticks1, ticks2, target;

This line creates three variables named ticks1, ticks2 and target. All three are of a special data type named clock_t. Now let's see how these variables are used. The next line is:

ticks1 = clock(), ticks2 = ticks1;

Here we have a new function, clock(), which gets the time in system clock ticks since the program has started. When clock() is called, the number of clock ticks is stored in the variables named ticks1 and ticks2. Then we have this line:

target = ms * CLOCKS_PER_SEC / 1000;

The variable named **ms** contains the number of milliseconds to pause. The constant named CLOCKS_PER_SEC is the number of times the computer clock ticks per second. Since we want to work with the unit millisecond, we need to divide the number of clock ticks per second by 1000 to get the number of clock ticks per millisecond. Then this value is multiplied by the number stored in **ms**, which is the desired number of milliseconds to pause [to be more correct, ms is first multiplied by CLOCKS_PER_SEC and THEN is divided by 1000]. The result of the calculation is stored in the variable named **target**. In other words, the variable named **target** stores a number which is equivalent to the number of computer clock ticks that happen in the number of milliseconds desired for the pause.

Then we finish with these lines:

```
while ( ( ticks2 - ticks1) < target)

    ticks2 = clock();
```

The while loop will run until the value stored in ticks1, subtracted from the value stored in ticks2, is no longer less than the value stored in target. As the loop runs, the value of ticks2 continues to increase because that value is assigned by the clock() function. As the clock is ticking, ticks2

gets larger, but ticks1 does not. So when ticks2 is larger than ticks1, by an amount of clock ticks that is equivalent to the desired pause time, then the while loop stops execution. At that point we need to look back in the main() code block on the line after pause(pause_time), where we find this line: printf("The pause has finished!\n\n");

I admit that the logic used in the program can be difficult to follow. What I would like you to take away from the lesson is this: **1)** it is possible to create your own custom functions **2)** when you create a custom function, you must announce it before the line containing the main() function and **3)** the code that defines the custom function is placed after the code block for the main() function.

Suppose that after you are more experienced in writing code, you find that you are using some of the same sets of code lines over and over. In that case maybe you should create a custom function. When you become more advanced, you can even learn how to make your own custom libraries to store all of your custom functions.

A custom function for the robot

Let's take one more example to finish off this lesson. In this case we will be using code that only works with the IDE for the robot, so you can't test it out with Dev-C++. The code defining the function is listed in the box below.

```
int ping_mm(int pin)
{
  return ping(pin) * 10 / 58;
}
```

The code above defines a custom function named **ping_mm()**. In the IDE for the ActivityBot robot, there is a library named ping.h . This library has functions that work with an ultrasonic distance sensor. In the ping.h library there is a function named **ping_cm()**, which returns the distance from the sensor to an object in units of centimeters. Later on, when you are working with the robot, you will need to write some code for the

ultrasonic senor that determines the distance in millimeters. Unfortunately, there is no **ping_mm()** function in the ping.h library. However, there is a function in that library that can be used to make a custom function named **ping_mm()**. So let's look at the code that creates this custom function. The first line is:

```
int ping_mm(int pin)
```

This line lets us know that the function will return a number of the integer type (int). The function name will be **ping_mm()**. The function takes one argument, an integer with the variable name of pin. In this case the pin is the input/output pin of the robot microcontroller that is connected to the signal pin of the sensor. There is only one more line of code:

```
return ping(pin) * 10 / 58;
```

There is a function in the ping.h library named **ping()**. This function returns the number of clock ticks between the time an ultrasound pulse is sent out and the time when it is detected after bouncing off an object. The robot clock ticks one million times a second or once each microsecond. If we know the speed of sound, then we can calculate the distance between the ultrasound sensor and the object.

At 20^{o}C and at standard air pressure at sea level, sound travels about 0.343 mm per microsecond or 2 mm in 5.8 microseconds. We want the figure in units of 2 mm because the distance must be doubled due to sound traveling out and back before detection. If the sensor is 1 mm from the object, a pulse of sound will travel out and back to the sensor in 5.8 microseconds. Now we need to multiply 5.8 microseconds (same as 5.8 clock ticks) by a number whereby the result will be one. If we multiply 5.8 by 0.17, the answer will be one. Also note that the fraction 10/58 equals 0.17. So this is how it works: ping() returns the number of clock ticks between when the sound is sent out and returns. Each tick equals one microsecond. If we multiply the number of clock ticks by 0.17, then we will have the distance in millimeters. So why do we first multiply by 10 and then divide by 58 instead of just multiplying by 0.17? We are using integers in our code because the function ping() returns integer numbers. We should not be multiplying with a floating point number like 0.17. Instead, we must first multiply the number of clock ticks by 10 and then divide that answer by 58. That is what this code does: `return ping(pin) * 10 / 58;`