CEENBoT-API Programming Fundamentals

Quick CEENBoT-API Programming Examples
CEENBoT v2.21 – '324 Platform

written by **Jose Santos,**
CEENBoT-API Creator and Developer

Current as of **`v1.09.000R`**
Rev 1.02

University of Nebraska, Lincoln (Omaha Campus)

(Blank)

Introduction

About This Document

The purpose of this document is to provide some fundamental examples with some common tasks that can be achieved with the CEENBoT-API, which includes such fundamental tasks as *delays*, *motion control*, and environment sensing via the CEENBoT's infra-red *bump* sensors. The goal of this document is to simply point you to some key API functions you'll want to immediately get to know when writing CEENBoT-API programs.

Prerequisites

This document assumes the following:

- You have a basic working knowledge of programming in C. Teaching you about the C programming language is beyond the scope of this document.
- **You have successfully read through to completion the *CEENBoT-API: Getting Started* manual.** This is *very* important.

  The manual discusses how to set up your *programming development environment/tools* (i.e., *AVR Studio 4*), takes you through the steps of *compiling* a CEENBoT-API program, and the steps required to *upload* it to your CEENBoT. Users are expected to have gone through this procedure.

  The *Getting Started* manual, along with other CEENBoT-API documentation can be found at:

  http://ceenbot.digital-brain.info

- You have the necessary hardware: *Windows-based* PC, CEENBoT, and AVR-ISP programmer.

Implementing *Delays*

One of the tasks for the CEENBoT programming exercise requires that you implement a *delay* into your program. A *delay* will cause your program to "pause" for a given amount of time (which *you* specify) before program execution continues.

For example, you may wish for your CEENBoT to *delay* for 5 seconds after power up, to give you time to set your CEENBoT down on the floor before its wheels start spinning.

There's an API function called `TMRSRVC_delay()`, which allows you to do *just* that. The function takes a single argument (or value) which specifies *how long* you wish to delay. For example, if you wish to wait 5 seconds, then you invoke the function in your program as:

```
TMRSRVC_delay( 5000 );
```

Why **`5000`**? Because the value given to the function is in units of *milliseconds* (NOT *seconds*). Therefore, **`5 seconds = 5 * 1000 = 5000 milliseconds`**.

Here's a super-simple example, on how you can use the delay function. Don't worry about everything else – the main point here is to convey an the idea:

```
// EXAMPLE 1:
#include "capi324v221.h"

void CBOT_main( void )
{

    // Open the LCD for use (so we can print messages).
    LCD_open();

    // Display a message.
    LCD_printf( "Hello, Dolly!\n" );

    // Wait 2 seconds.
    TMRSRVC_delay( 2000 );

    // Display another message.
    LCD_printf( "How are you?\n" );

    // Infinite loop.
    while( 1 );

} // end main()
```

This program starts by printing the message **`"Hello, Dolly"`** on the display. Then, *nothing* is going to happen for **`2 seconds`** (**`2000 milliseconds`**). After that, another message prints, this time being **`"How are you?"`**. After that, the program gets stuck in an infinite loop.

You can insert delays almost anywhere in your program and anytime you wish your CEENBoT to ***wait*** for some time before it does the next thing. For example, you could have your CEENBoT start by moving forward, and then, wait 5 seconds, and then move back, or turn, etc. The *delay* function allows you to do things like that. The only *caveat* is that the *maximum* delay possible is **`30 seconds`** (or **`30000 milliseconds`**) – if you want longer delays than that, then you can call `TMRSRVC_delay()` multiple times back-to-back, or by using one of C's several *looping* constructs.

Moving the CEENBoT

Fundamental Concepts

The next *obvious* fundamental task you will be required to do is to get your CEENBoT moving. The wheels of your CEENBoT are each attached to what is called a *stepper motor*. They're called that because the motors move forward (or back) in small incremental units called "steps".

The motors that are installed on the CEENBoT have **`200 steps`** for one revolution, so that each step rotates the wheel **`1.8-degrees`** (that's **`360/200 = 1.8`**).

The reason you need to be aware of this "stepping" system is because you can't tell the CEENBoT to move it's wheels, say **`5-ft,`** or **`10-inches`**, or **`2-cm`**. The CEENBoT doesn't understand those units – the CEENBoT only understands "steps". So to get your CEENBoT to travel a certain distance, you have to tell it ***how many steps*** it should rotate its wheels instead. Turn to the next page to read about fundamental *motion* functions.

Supplementary Note – It is possible to create a relation between "steps" and "distance". Recall the *circumference formula*:

$$C = 2\pi r$$

If you measure the *radius* r of the wheel *very* carefully, you can compute the circumference C. Then, you can compute *how far* each "step" moves you (i.e., the *distance-per-step*) by using the conversion:

$$d_{\text{step}} = \frac{C}{200}$$

Finally, if D represents the total distance you want to travel, the "number of steps" required to travel that distance can be computed as:

$$N_{\text{steps}} = \frac{D}{d_{\text{step}}}$$

The *Motion* Functions

The CEENBoT-API offers a dozen or so functions that allow you to get your CEENBoT to move in a variety of ways. However, this 'multitude' of functions in the API reference documentation manual can be a bit overwhelming the first time around. It turns out that most of the *motion* tasks you can perform with your CEENBoT can be accomplished with the following functions:

- **`STEPPER_move_stwt()`** (the *step-and-wait* function)
- **`STEPPER_move_stnb()`** (the *step-no-block* function)
- `STEPPER_stop()` (can only be used in conjunction with `STEPPER_move_stnb()`)
- `STEPPER_wait_on()` (can only be used in conjunction with `STEPPER_move_stnb()`)

The first two motion functions are the most important – these are the ones that cause the CEENBoT wheels to move. These two functions allow you to *independently* control the left and right wheels/motors and set the *direction*, *speed* and *acceleration* of each.

Here's our first example – it causes the CEENBoT to move forward a finite distance, then turn, and move forward some more before coming to a stop. Study the following code sample carefully – the *key* lines are numbered in parenthesis for the discussion that follows on the next page:

```
// EXAMPLE 2:
#include "capi324v221.h"

void CBOT_main( void )
{

    STEPPER_open();     // Open STEPPER module for use.              (1)

    // Move BOTH wheels forward.
    STEPPER_move_stwt( STEPPER_BOTH,                                 (2)
        STEPPER_FWD, 1000, 200, 400, STEPPER_BRK_OFF,   // Left
        STEPPER_FWD, 1000, 200, 400, STEPPER_BRK_OFF ); // Right

    // Then TURN RIGHT (~90-degrees)...
    STEPPER_move_stwt( STEPPER_BOTH,                                 (3)
        STEPPER_FWD, 150, 200, 400, STEPPER_BRK_OFF,   // Left
        STEPPER_REV, 150, 200, 400, STEPPER_BRK_OFF ); // Right

    // Move BOTH wheels forward.
    STEPPER_move_stwt( STEPPER_BOTH,                                 (4)
        STEPPER_FWD, 1000, 200, 400, STEPPER_BRK_OFF,   // Left
        STEPPER_FWD, 1000, 200, 400, STEPPER_BRK_OFF ); // Right

    // Infinite loop!
    while( 1 );                                                      (5)

}
```

In line (1) we start up the *module* that contains all the stepper-related functions. This is required.

Then in line (2) we initiate a motion. Note that this is a *single line*, with its parameter values spread over 3 consecutive lines separated by commas – so keep in mind this constitutes a single "function call", which I've replicated below:

```
STEPPER_move_stwt( STEPPER_BOTH,
        STEPPER_FWD, 1000, 200, 400, STEPPER_BRK_OFF,   // Left
        STEPPER_FWD, 1000, 200, 400, STEPPER_BRK_OFF ); // Right
```

`STEPPER_BOTH` indicates that the parameters for *both* the left and right motors are valid. Sometimes you only want to 'affect' a *single* motor, but you still have supply values for both left and right. In this case we can specify `STEPPER_LEFT`, or `STEPPER_RIGHT`, but here we want *both* motors to be affected, so we use `STEPPER_BOTH`.

Then, what follows are the *parameters* for the LEFT and RIGHT motors (one on each line). The first parameter represents the *direction* the wheel is to move: `STEPPER_FWD` (move forward), or `STEPPER_REV` (move in reverse). Then, the *distance* the wheel is to move – here we specified **`1000 steps`**. This is followed by the *speed* the motor is to move – here we specified **`200 steps-per-second`**. After that, we specify the *acceleration* of **`400 steps/sec`**^{2}. Finally, the *brake mode* – `STEPPER_BRK_OFF` means we want to keep the motor "brakes" *dis-engaged* once the motion completes – that is, after the wheel has turned all **`1000 steps`**. If you want to engage the brakes, you would specify `STEPPER_BRK_ON`.

So in summary, the line above (which represents line (2) on our program sample) says to move *both* wheels forward, for **`1000`** steps at a speed of **`200`** steps-per-second and acceleration of **`400 steps/sec`**^{2}; furthermore, we want the brakes off when this command completes.

Lines (3) and (4) perform the same task. The only difference is that in line (3) we *turn right* by making the left wheel move forward, and the right wheel move in reverse – also note that each wheel is moving for 150 steps. This gives me approximately a 90-degree turn, but your results may vary.

```
STEPPER_move_stwt( STEPPER_BOTH,
        STEPPER_FWD, 150, 200, 400, STEPPER_BRK_OFF,   // Left
        STEPPER_REV, 150, 200, 400, STEPPER_BRK_OFF ); // Right
```

Then, in line (4), we move forward again for the *same* distance (1000 steps), same speed, and same acceleration.

```
STEPPER_move_stwt( STEPPER_BOTH,
        STEPPER_FWD, 1000, 200, 400, STEPPER_BRK_OFF,   // Left
        STEPPER_FWD, 1000, 200, 400, STEPPER_BRK_OFF ); // Right
```

Note that lines (2), (3) and (4) execute sequentially. Line (3) will NOT execute until line (2) finishes – that is, until all 1000 steps have occurred. Line (4) will NOT execute until line (3) finishes – that is, until all 150 steps have occurred, etc. This is why `STEPPER_move_stwt()` is called the *step-and-wait* function. It instructs the wheels to step a given amount, and the program will *wait* until this motion completes. Make sure you keep this idea in mind because the next example shows the alternative scenario – what if, instead of *step-and-wait*, we just *step-and -**don't**-wait*?

Let's look at the example on the next page to see how we can achieve this scenario:

This next example shows how you would use the `STEPPER_move_stnb()` function (this is the *step-no-block* function). Imagine that we need to get the CEENBoT moving, but AS SOON as you get the robot moving, you need to begin "scanning" for possible objects in the way?

You can't do this with *step-and-wait* because these functions DO NOT let you do anything else until the motion completes. We need an alternative...

`STEPPER_move_stnb()` is precisely for this kind of scenario.

```
// Example 3:
#include "capi324v221.h"

void CBOT_main( void )
{

    STEPPER_open();     // Open STEPPER module for use.

    // Move forward.
    STEPPER_move_stnb( STEPPER_BOTH,                                    (1)
        STEPPER_FWD, 5000, 200, 450, STEPPER_BRK_OFF,   // Left
        STEPPER_FWD, 5000, 200, 450, STEPPER_BRK_OFF ); // Right

    // <<< ...do something else here... >>>                             (2)

    // Wait **HERE** for motion to complete on BOTH motors
    // before we do anything else.
    STEPPER_wait_on( STEPPER_BOTH );                                    (3)

    // Move back.
    STEPPER_move_stnb( STEPPER_BOTH,                                    (4)
        STEPPER_REV, 5000, 200, 450, STEPPER_BRK_OFF,   // Left
        STEPPER_REV, 5000, 200, 450, STEPPER_BRK_OFF ); // Rev.

    // <<< ... maybe do something else here... >>>                      (5)

    // Wait **HERE** for motion to complete on BOTH motors
    // before we do anything else.
    STEPPER_wait_on( STEPPER_BOTH );                                    (6)

    // Infinite loop!
    while( 1 );

}
```

Line (1) starts the motion. Both steppers move forward, for 5000 steps at a speed of 200, and acceleration of 450. We also state that the brakes should stay OFF when motion completes. Now, unlike `STEPPER_move_stwt()` (*step-and-wait*), here, we're using the *step-no-block* version. This means, as soon as the wheels begin to move, execution moves to *whatever* you may have after that. This "whatever" is indicated in line (2).

Then comes an important function. `STEPPER_wait_on()`. This function is used to hold execution of the program to continue any further, until either the LEFT, RIGHT or BOTH stepper motors complete their motion in their entirety. We need this to prevent line (4) from starting while the motors are *still* trying to finish based on what was instructed in line (1).

When the motion completes, then execution moves to line (4), and now the motors begin to move backward. IMMEDIATELY after that, whatever code you have in line (5) gets executed also.

We then use line (6) to *hold off* again, and wait for the *previous* motion to complete.

The program concludes by entering the *infinite* `while()` loop.

Reacting to the Bump Sensors

The CEENBoT comes equipped with two forward Infra-Red (IR) *bump* sensors. The CEENBoT-API provides a function to "query" the state of these sensor to determine if an object is blocking a sensor or not. The user can then use the state obtained from this query to take the appropriate action (e.g., *if left blocked, then go around* object), etc. The *key* function that allows us to do this (query bump sensor state) is the **ATTTINY_get_IR_state()** as showcased in the example below that follows. This function returns 'TRUE' (a *non-zero* value), if the state of the requested IR sensor is active (i.e., being *blocked*).

The following program causes the CEENBoT to turn right 90-degrees if the right bump sensor is triggered, and left 90-degrees if the left bump sensor is triggered.

```
// Example 4:
#include "capi324v221.h"

void CBOT_main( void )
{

    STEPPER_open();

    // We do this repeatedly for ever.
    while( 1 )
    {

        // Wait a bit...
        TMRSRVC_delay( 125 );

        // If LEFT sensor is triggered then move LEFT 90-degrees.
        if ( ATTINY_get_IR_state( ATTINY_IR_LEFT ) == TRUE )          (1)
        {
            // Turn LEFT...
            STEPPER_move_stwt( STEPPER_BOTH,
                STEPPER_REV, 150, 200, 400, STEPPER_BRK_OFF,    // Left
                STEPPER_FWD, 150, 200, 400, STEPPER_BRK_OFF );  // Right

        }

        // Otherwise, if the RIGHT sensor is triggered, then move RIGHT 90-degrees.
        else if ( ATTINY_get_IR_state( ATTINY_IR_RIGHT ) == TRUE )    (2)
        {
            // Turn RIGHT...
            STEPPER_move_stwt( STEPPER_BOTH,
                STEPPER_FWD, 150, 200, 400, STEPPER_BRK_OFF,    // Left
                STEPPER_REV, 150, 200, 400, STEPPER_BRK_OFF );  // Right

        }

    } // end while()

} // end CBOT_main()
```

The first thing to notice about this program is that except for the first line, the entire program is inside of a *while* loop, which will execute repeatedly forever.

At the beginning of the *while* loop we introduce a little "delay" to give the CEENBoT time to respond to the sensor requests afterwards. This also allows us to control how quickly the loop runs – a **`125ms`** delay means the contents in the while loop will execute approximately 8 times per second.

In the line labeled (1), we call the `ATTINY_get_IR_sensor()` function. This function reads the state of the bump sensors and returns a *boolean* value of type `BOOL`, which will be equal to `TRUE` (a *non-zero* value), if an object happens to be blocking the IR sensor, and `FALSE` (a *zero* value) otherwise.

Therefore, going back to (1) in the example, if the state of the *left* IR sensor is active (`TRUE`), then the code block immediately beneath the `if()` statement is executed. We use the `STEPPER_mov_stwt()` (this is the *step-and-wait* versions of motion functions) to move the robot to the left in a tank-like fashion.

If the *left* IR sensor is not triggered, we repeat the process and check if the *right* IR sensor is active (`TRUE`). If so, then the code immediately beneath the `if-else()` block is executed, forcing the robot to turn *right*.

If neither the left or right IR sensors are triggered, then nothing happens, and everything repeats again on the next loop iteration – whereby we continuously check the left OR right sensors for activity. If something happens, then we react accordingly; but if nothing happens, we just keep checking until something *does* happen.

Again, notice that we are using the *step-and-wait* functions to move the robot. After the motion completes, the program starts again at the beginning of the *while* loop, where this same process repeats.