CEENBoT-API Programming Fundamentals

Quick CEENBoT-API Programming Examples
CEENBoT v2.2x – '324 Platform

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

Current as of **`v2.04.128R`**
Rev 1.12

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. These include implementation of *delays*, *motion control*, and environment sensing via the CEENBoT's infra-red *bump* sensors. It also shows basic usage of the LCD display, LEDs and querying the state of the push-buttons. 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 given it can be overwhelming if you dive straight into the 270 page *CEENBoT-API Programmer's Reference.* Further more the 'reference' is just that, a 'reference', not a tutorial. Now, it should be stated that this document is not so much a tutorial but a way to give you a better "feel" for the things you can do, and the things you need to know how to do with the CEENBoT right away. Having said that, it is important as you read over this document to have access to the *CEENBoT-API Programmer's Reference Guide* which contains all the technical details that are merely touched upon "on the surface" in this document.

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. *NOT knowing* how to program in C is the biggest hurdle in doing CEENBoT-API programming with your CEENBoT.

- **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* or *Atmel Studio 6.x*), 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. Thus, it will be assumed you know how to create a CEENBoT-API project and be able to compile and generate the HEX file that is uploaded to the CEENBoT. This guide is about programming features of the API, not the tools used in the process.

  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.

- You have access to the *CEENBoT-API Programmer's Reference Guide*. This is the go-to guide to understand all the details behind each of the API functions and their associated arguments. It contains EVERYTHING you need to know about EVERYTHING you can do with the API once you have gained some experience on writing API code, this will be *the* guide to go to.

Let's get started.

Writing to the LCD (CEENBoT's display)

Now that you have the fundamental "ingredients" of a CEENBoT-API program, we'll start with the most basic task – writing to the CEENBoT's on-board display or LCD.

This tutorial shows you how to display messages on the CEENBoT's LCD. The CEENBoT comes equipped with a 128x64 pixel liquid crystal display. In it, you can write *four* lines with at most 21 characters on each line (21x4), so when you display messages, you have to make sure your *sentences* (*string length)* do not exceed 21 characters or otherwise they'll wrap around to the next line.

Here's a *very* basic program showing how to do that – note how all the required *elements* previously discussed are all there.

```
#include "capi324v221.h"                        (1)

void CBOT_main( void )                          (2)
{

    // Start up the LCD.
    LCD_open();                                 (3)

    // Display some messages.
    LCD_printf( "Hello, Dolly!\n" );            (4)
    LCD_printf( "CEENBoT is ready!\n" );        (5)

    // Don't leave.
    while( 1 );                                 (6)

} // end CBOT_main()
```

Line (1) is our *required header* file of any CEENBoT-API program. It contains all symbolic constants and declarations needed to successfully compile your CEENBoT-API code.

Line (2) is the *entry* point (e.g., starting point) of all CEENBoT-API programs – `CBOT_main()`.

Now comes the interesting part: line (3). Line (3) is like an ON switch to your car's *radio*, or *CD player*. The CEENBoT is no different. You want to use the LCD display? Then you *have* to turn it ON first. You do this by calling `LCD_open()`, which "opens" and initializes resources specific to the LCD and makes them available to *you*, the user. Many of the CEENBoT's on-board peripherals use this "*open-before-use"* approach.

> **Note:** Always consult the "*CEENBoT-API: Programmer's Reference Guide*" to determine if a specific on-board peripheral needs to be opened, or if it is already open by default.

After the LCD is *opened* we can go ahead and use it. The simplest way to display messages is via the `LCD_printf()` function as shown in lines (4) and (5). `LCD_printf()` works pretty much like your traditional `printf()` function in C, and just like it, it allows you to *output* text messages to the *display device*. In our case, this *display device* is the LCD.

Note that text strings are enclosed in *double-quotes*. Also note the *escape character* '**\n**', which denotes a *new line* or *carriage return*. It causes the cursor to move to the "next line down", *just like in* `printf()`.

Finally, line (6) is that *infinite loop...* which keeps the “engine running” and prevents program execution to leave the CEENBoT's execution environment.

Other Basic LCD Operations

The `LCD_clear()` function will clear the contents of the LCD display, and anything you write after that by issuing `LCD_printf()` will start at the top of the LCD. Note that `LCD_open()` automatically clears the LCD, so there's no need to follow an `LCD_open()` with an `LCD_clear()`.

Another task is the ability to write to a specific line (or *row*) on the LCD. The LCD has four lines (or rows) you can write to. You can write to a specific line by using the `LCD_printf_RC()` function (The “RC” stands for *row-column*). It has the following prototype:

```
LCD_printf_RC( row, column, string_format, ... );
```

The *row* value is an integer between `0` and `3`, with `3` being the TOP-MOST line of the LCD, and `0` being the BOTTOM-MOST line on the LCD. The *column* is actually the *pixel* value to start. This allows you to offset characters in pixels instead of characters, but for most purposes the *column* value should be `0`, so strings are printed starting at the very beginning of each line of the LCD. Thus, to print a string on the first and second lines you would write:

```
LCD_clear();                                // Clear the display.
LCD_printf_RC( 3, 0, “First line!\t” );     // Print specifically on first line.
LCD_printf_RC( 2, 0, “Second line!\t” );    // Print specifically on second line.
```

The `\t` in the string is actually *not* a TAB, but instead causes everything that follows the line to be erased. This is useful in *loops* where you consistently write strings to the same specific line of different lengths. If in one instance your string is long, and on another instance you overwrite the same line with a string that is shorter, then you will still see characters from the previous string. To avoid this, you always want to erase everything that follows on that one line.

```
unsigned int i = 0;

for ( i = 0; i < 10; ++i )
{

    // Always print on the first line of the LCD.
    LCD_printf_RC( 3, 0, “i = %d\t”, i );

    // Wait 1 second.
    TMRSRVC_delay_sec( 1 );

}
```

This approach is always better than doing this, which you should omit as much as possible (see next page):

```
unsigned short int i = 0;

for( i = 0; i < 10; ++i )
{

    // Clear contents first.
    LCD_clear();
    LCD_printf_RC( 3, 0, "i = %d', i );

    // Wait 1 second.
    TMRSRVC_delay_sec( 1 );

}
```

Invoking `LCD_clear()` takes a long time because you're clearing the whole display. In fact, if you're clearing the whole display, then using `LCD_printf_RC()` might be pointless because the purpose behind writing to a specific line is that you may have something previously written on another. `LCD_printf_RC()` allows you to modify some portions of the LCD while keeping existing content on other portions intact.

Let me give you one final example before moving on. Suppose you wrote a program where the CEENBoT does different things, so you use the display to denote 'what state' the CEENBoT is on. So on the top-most line, you print "**`State`**" and on the line below it, you print that state. Suppose this line can have: "**`Idle`**", "**`Running`**", "**`Sensing`**", "**`Avoiding`**", so you want your display, for example, to say:

```
State
Running
```

when the robot is running, so you use `LCD_printf_RC()` to print to the first line, and again to print on the second line. Now suppose the state changes to "Idle", so you want to print that on the second line. So you would, say, issue `LCD_printf_RC( 2, 0, "Idle" )` without the **`\t`**. Then, this would result in:

```
State
Idleing
```

Because the new string is shorter, your string did not completely cover or overwrite the previous string, so you see "**`Idleing`**" because only the four first characters were overwritten in "**`Running`**". However, if you use the \t and instead issue `LCD_printf_RC( 2, 0, "Idle\t" )` then you'd get:

```
State
Idle
```

which is what you want.

Lastly, you may be wondering, *"why did you use \t? That's for TAB!"*. That's true. When the CEENBoT-API was developed from prototype work from an existing firmware that had been written by someone else, it was decided to respect these "odd" decisions and kept this as it is now. :)

Strings Can Eat Up SRAM

Okay, so now you know some of the basic LCD functions you can use to display information, but there's one minor caveat you should be aware of. Strings consume SRAM. The more strings you have the more memory you use up. Since the ATmega324 MCU on the CEENBoT doesn't have a lot of SRAM, you need to be mindful of the number of strings you have within your code. For example, doing:

```
LCD_printf( "The big brown fast fox jumped over the slow lazy dog!" );
```

occupies 54 bytes of SRAM (including the terminating NULL character). If you just have ten strings that are that long on average, you'll use up a half a kilobyte already.

To remedy this situation, the CEENBoT-API has *program memory* versions of the `LCD_printf()` functions, which automatically store the strings in program memory. Since you have far more program memory than you do SRAM (32K vs. 2K SRAM), it is a good idea to always store your strings in program memory. Two functions we've discussed thus far are `LCD_printf()` and `LCD_printf_RC()`. The program memory versions of these are called `LCD_printf_PGM()` and `LCD_printf_RC_PGM()`. You must use these functions in conjunction with a `PSTR()` macro as shown in the following examples:

```
LCD_printf( "Hello, Dolly!" );                // Stores 14 bytes in SRAM.
```

versus...

```
LCD_printf_PGM( PSTR( "Hello, Dolly!" ) );    // Stores 14 bytes in Program Memory.
```

The `PSTR()` macro must enclose any string. This tells the compiler to store the string in program memory (or that the source of the string is *from* program memory). Here's another example:

```
unsigned short int month = 12;
unsigned short int day   = 25;

// String is stored in SRAM.
LCD_printf( "Presents arrive on %d/%d, Christmas day!", month, day );

// Same string, but now stored in Program Memory.
LCD_printf_PGM( PSTR( "Presents arrive on %d/%d, Christmas day!" ), month, day );
```

Again, note that the `PSTR()` macro only encloses the string.

Here's another example when we must write to specific row/column:

```
// Print message in second line from the top.
LCD_printf_RC( 2, 0, "Hello, Dolly!\t" );                    // SRAM.

// Print message in second line from the top.
LCD_printf_RC_PGM( 2, 0, PSTR( "Hello, Dolly!\t" ) );        // Program Memory.
```

Again, pay careful attention, how `PSTR()` only encloses the string.

So, in summary. If you only have a few short strings. You can just invoke the basic functions that store strings in SRAM. There's NOTHING wrong with this, since these are less verbose. But if you have a LOT of strings, you should most definitively store them in Program Memory, so you don't end up eating your SRAM simply because of your strings.

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.

There's also a `TMRSRVC_delay_sec()` function that takes a delay in units of *seconds*. So, in the previous example, you could have said `TMRSRVC_delay_sec( 2 )` instead of `TMRSRVC_delay( 2000 )`. Both achieve the same thing.

The LEDs

The CEENBoT comes equipped with two on-board LEDs (Light-Emitting Diodes). One is *green* and the other is *red*. There is also a third green LED beneath the LCD display, next to the push-buttons, but this LED is not accessible by the user. In any case, the following program shows how to manipulate the LEDs.

```
#include "capi324v221.h"

void CBOT_main( void )
{

    int i = 0;                                  (1)

    // Open the LEDs for use.
    LED_open();                                 (2)

    // Turn the Green LED ON.
    LED_set( LED_GREEN );                       (3)

    for( i = 0; i < 10; ++i )                   (4)
    {

        // Toggle the Red LED.
        LED_toggle( LED_RED );                  (5)

        // Wait 250 milliseconds before looping again...
        TMRSRVC_delay( 250 );                   (6)

    } // end for()

    // Turn the Green LED OFF.
    LED_clr( LED_GREEN );                       (7)

    // Wait forever...
    while( 1 );                                 (8)

} // end CBOT_main()
```

As before we need to initialize the LED subsystem module by invoking `LED_open()`. Then we turn an LED ON using `LED_set()` macro-function, and turn it off by using the `LCD_clr()` macro-function, with the constants `LED_RED`, or `LED_GREEN` to decide which LED is affected. The CEENBoT only has two such LEDs.

Another macro-function exists for simply *toggling* the state of the LED (i.e., if ON, turn OFF; if OFF, turn ON). This is the `LED_toggle()` macro-function, whose only argument specifies *which* LED will be toggled. Again, the choices are `LED_RED` or `LED_GREEN`.

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: Moving the CEENBoT

The CEENBoT-API offers a dozen or so functions that allow you to get your CEENBoT to move in a variety of ways. In fact, there are a variety of ways to achieve the same desired motion because of the plethora of functions for moving the CEENBoT in the STEPPER subsystem module. 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. You should focus your motion tasks to making use of these functions because they're *high-level* motion functions. Refer to the *CEENBoT-API Programmer's Reference Guide* for details regarding these functions. In any case, the key functions are:

- **`STEPPER_move_rn()`** (the *run* [perpetually] function)
- **`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()`** or **`STEPPER_move_run()`**)
- `STEPPER_wait_on()` (can only be used in conjunction with **`STEPPER_move_stnb()`**)

The first three 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. These three parameters define the motion of each wheel.

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/sec`**. 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 and you will need to experiment for this especially if you have the rigid wheels, vs the rubbery ones.

```
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 (emphasis on "wait"). 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*? (e.g., step and immediately move to the next line in my program).

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 (just pretend with me here).

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). This could be one line of code, or a 100 lines of code.

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, be it one line or a 100.

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.

Last, but not least is the `STEPPER_move_rn()`. All the functions we've explored thus far are *finite-motion* commands. That is, in addition to direction, speed, and acceleration parameters, we state how many steps each wheel should move (e.g., we define distance). This means, when all the wheels have moved the number of steps given, the wheels *will* stop. With `STEPPER_move_rn()`, however, you don't specify the number of steps (distance does not play a role). This means the wheels will continue to spin at the direction, speed, and acceleration given until you issue a `STEPPER_stop()` (or until your battery runs out of juice, whichever happens first). This function is also *non-blocking*.

The following sample shows how to use this function by having the CEENBoT spin in place for three seconds, and then stopping it.

```
#include "capi324v221.h"

void CBOT_main( void )
{

    STEPPER_open();

    // Run perpetually...
    STEPPER_move_rn( STEPPER_BOTH,
        STEPPER_FWD, 200, 450, STEPPER_BRK_OFF,     // Left
        STEPPER_FWD, 200, 450, STEPPER_BRK_OFF );   // Right.

    // Wait 3 seconds.
    TMRSRVC_delay_sec( 3 );

    // Stop the on-going motion, keep brakes disengaged.
    STEPPER_stop( STEPPER_BOTH, STEPPER_BRK_OFF );

    // Infinite loop!
    while( 1 );

} // end CBOT_main()
```

Note the format for the arguments is almost exact as `STEPPER_move_stwt()` and `STEPPER_move_stnb()`. The only difference is that we do not say *how far* the wheels should move. Lastly, because the wheels will move perpetually, we have to decide when to force a stop, which we do via `STEPPER_stop()`. We did a little delay of 3 seconds to allow the robot to move this long before instructing it to come to a full stop.

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 sensors 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 sensor is blocked, then go around* object), etc. The key function that allows us to do this (query bump sensor state) is the **ATTINY_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*) and 'FALSE' otherwise. 'TRUE' and 'FALSE' are special Boolean constants defined in the API.

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. The example uses STEPPER_move_stwt() discussed previously.

```
// 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.

> **Note:** This `BOOL` type is defined by the API. It is not part of the C language. Also, do not confuse BOOL type defined in the API with `bool` that is part of the C++ standard.

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_move_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.

Using the Push-buttons

The CEENBoT has three general purpose push-buttons labeled "S3", "S4" and "S5" on the controller board beneath the LCD. User's can query the state of the push-buttons in a similar manner to querying the state of the IR sensors. This can be done via the `ATTINY_get_SW_state()` function.

The following sample program shows how to toggle the LEDs by using the push-buttons using the `ATTINY_get_SW_state()` function (*see next page*):

```
#include "capi324v221.h"

void CBOT_main( void )
{

    LED_open();

    while( 1 )
    {

        if ( ATTINY_get_SW_state( ATTINY_SW3 ) == TRUE )
        {

            // Toggle the LED.
            LED_toggle( LED_GREEN );

            // De-bounce the button.
            TMRSRVC_delay( 100 );

        } // end if()

        if ( ATTINY_get_SW_state( ATTINY_SW4 ) == TRUE )
        {

            // Toggle the LED.
            LED_toggle( LED_RED );

            // De-bounce the button.
            TMRSRVC_delay( 100 );

        } // end if()

    } // end while()

} // end CBOT_main()
```

Note that while there's an `LED_open()`, there's *no* `ATTINY_open()`. This is because the ATTINY subsystem module is open by default. The `ATTINY_get_SW_state()` returns a boolean value previously discussed to be `TRUE`, if the button has been depressed and `FALSE` otherwise. It takes a single argument, which is the push-button whose state you wish to check. Three constants have been defined for specifying this argument: `ATTINY_SW3`, `ATTINY_SW4`, and `ATTINY_SW5`.

The Timer Service & Timer Objects

Another sophisticated component of the CEENBoT-API is the ability to create what are called *timer objects*. Timer objects can be created so that they "trip" after a certain amount of time has passed (I suppose a better description is an "clock alarm"). If the timer object has also been configured to automatically "restart" again, then it will do so and "trip" again after its programmed time has elapsed. Timer objects are useful in sophisticated timing scenarios where a simple `TMRSRVC_delay()` won't do. For example, toggling an LED On and Off every second can be easily accomplished as follows:

```
while( 1 )
{

      // Toggle the Red LED.
      LED_toggle( LED_RED );

      // Wait a bit before looping.
      TMRSRVC_delay_sec( 1 );

}
```

The above works, so long you don't have to do anything else while you *wait*.

The problem is that sometimes you do need to do other stuff *while* you wait. Suppose your robot needs to sample data from its sensors while you wait for "1 second" to elapse. Simply 'delaying' wastes time and your program can't do anything else until the delay has fully elapsed. One second is a LOT of time in computer time. ;)

This is where timer objects come in handy. You can create a timer object to internally 'tick' once every second. While we wait for this timer to let us know when this second has elapsed, we can do other stuff and we don't have to "wait". This is how you would do it (*see next page, and then come back*).

The program starts by declaring a timer object, whose type is `TIMEROBJ` called `led_timer`. This so-called "object" is a structure, which has a field called **`tc`** for *terminal count*. This field gets set to "1" (like a flag) when the timer's pre-programmed timer interval has fully elapsed. The timer is then started by creating the object in memory via `TMRSRVC_new()` function call. In this function call we specify the timer by passing the address of our timer object. We specify that we want to know when the time has "elapsed" by way of a flag, which we do with `TMRFLG_NOTIFY` argument. We also specify that when the time has elapsed to automatically "restart" (via `TMR_TCM_RESTART`) and that the timer is to "trip" every 1000 milliseconds (1 whole second).

After that, we enter the infinite *while* loop. While we're looping we do something complicated. Immediately after that we "check" if the timer's flag has been set, by using the `TIMER_ALARM()` macro, which will return a non-zero value if it's set, and zero otherwise. If it is non-zero (which is interpreted as 'true') that means a whole second has passed by and its time for us to toggle the LED. If it hasn't we simply do nothing and loop again to do our complicated task. The point being, that we don't *waste time* waiting for a whole second to elapse. Instead, we can be more efficient with our program by doing something more important while we wait for the timer's time to elapse. This is better than simply using a long "delay". Note at the end of the `if()` block, we indicate we want to wait for another second to elapse and be notified of this condition. We do this via `TIMER_SNOOZE()` macro, which is like "clearing the alarm" , but with the understanding the alarm will "sound again". ;)

```
#include "capi324v221.h"

void CBOT_main( void )
{

    TIMEROBJ  led_timer;     // Declare a timer object.

    LED_open();

    // Create and start a timer to 'tick' (internally) every second.
    // The 'times_up' flag will be set to 1 each time this happens.
    TMRSRVC_new( &led_timer, TMRFLG_FLAGNOTIFY, TMR_TCM_RESTART, 1000 );

    // Here we would do something complicated inside of 'process_sensors()',
    // while keeping an eye on the timer flag.  When the flag gets set, this
    // means a whole second has passed and you can consider yourself 'notified'.
    while( 1 )
    {

        // Do something complicated.
        process_sensors();

        // Check if the time is up.
        if ( TIMER_ALARM( led_timer ) )
        {

            // Toggle the LED.
            LED_toggle( LED_GREEN );

            // Clear the flag so we see when it trips again.
            TIMER_SNOOZE( led_timer );

        } // end if()
    }

} // end CBOT_main()
```

It is important that you don't abuse this feature. Timer objects are memory hungry, and creating (and especially destroying) a timer is *expensive*. Timers should be created once and have them exist perpetually for the duration of the program. It is very bad programming practice to create and destroy a timer by invoking `TMRSRVC_new()` inside of a loop. This will consume all your SRAM and crash your program, so you need to be thoughtful about how to solve your timer needs with as few timers as possible.

For example, if you need 5 timers that each tick at intervals of one second, you don't create five timers. Instead, you create *one* timer that ticks one second, and then declare integer variables to represent each of your counters. When the timer object says a second has elapsed, then you increment each of the variables. This way each variables keeps a different time, but all ticking to the beat of the ONE timer object.

What Next?

Play! Play with your CEENBoT. Write little snippet programs to test your understanding of the API, the robot, and the C programming language. Consult the *CEENBoT-API Programmer's Reference* to see all that the CEENBoT-API can do and to learn more about function details and their arguments. The API allows you to:

- Beep the CEENBoT's speaker to create musical sounds.
- User timer objects for sophisticated timing tasks, or use a specialized set of functions called *CRC Functions* (call-rate controlled) used frequently in loops.
- Use a Playstation controller to read information from it to control your robot with your API program.
- Interface various sensors via I2C, SPI, and UART interfaces.
- Sample analog voltages via the ADC subsystem.
- Hook up a CMUcam for color tracking (CMUcam4 or CMUcam5).

I hope you enjoy the CEENBoT as much as I enjoyed developing for it.