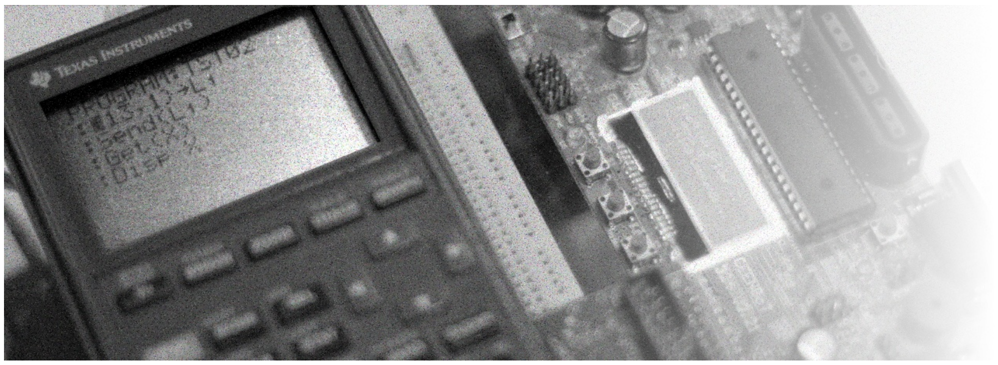
Robotics Solutions for Educators, Hobbyists and Researchers

CEENBoT / TI-Calculator User's Guide & Command Reference

User's Guide for using the TI-Calculator with the CEENBoT
CEENBoT v2.21 to v2.23 – '324 Platform



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

Department of Computer and Electronics Engineering (CEEN)
University of Nebraska-Lincoln (Omaha Campus)

Rev 1.04
(Current as of Driver: **v1.05.024R**)

STOP!

The intended audience of this document is for those wishing to try out and 'test drive' the TI-driver firmware to use your TI-calculator with your CEENBoT. It is intended for those with the capability and technical know-how to do so since this software is still in development and not yet 'production ready' for the 'masses'. It requires that you know how to program your CEENBoT (flash, or upload software) and have the resources (programmer, AVR studio software, and PC) for you to do so. If you're not comfortable with these procedures you're highly advised to wait for the 'production version' of the software.

Having said that, this document also contains information regarding the *command set* reference that is supported by the TI-driver firmware – thus, if this is all you're after, you can freely skip to *Chapter 2* of this document for that information.

NOTE

Some features may or may not be yet readily available and as development of the CEENBoT/TI integration continues, aspects of this document are subject to change at any time. Always check the source where you obtained this document to ensure you always have the latest revision of both, documentation and the driver software.

Document Conventions

This document uses the following typographical conventions:

- Code is written using `Lucida Console` type font. It is typically shown as follows:

```
Void CBOT_main( void )
{

    // ... code here ... ;

} // end CBOT_main()
```

or using the following:

```
void CBOT_main( void )
{

    // ... code here ...

} // end CBOT_main()
```

- Important details of technical interest (numerical values, bit field options, module names) are given in **`Courier New`** font. For example:

  "The **`STEPPER`** module requires the speed between **`0`** to **`400 steps/sec`**."

- Important notes or comments are given in *gray boxes* – for example:

Note: *Never* stick in your ear anything smaller than your elbow.

Comments, Questions, Document Errors and/or Suggestions...

Comments, questions and/or suggestions should be addressed by e-mail to:

`ceenbot.api@digital-brain.info`

Check out the *CEENBoT Portal* for latest development news regarding the API:

`http://ceenbot.digital-brain.info`

WARNING: Before You Begin...

Your CEENBoT may have arrived in your hands with a pre-programmed 'factory' firmware that showcases some basic functionality. More importantly, this functionality includes ***power management*** and ***battery charging*** capability. This capability is NOT yet included in the API. It is *very* important that you either have a backup, or have a copy of the original HEX file of the original CEENBoT factory firmware before you flash the *driver* HEX file that allows you to use the CEENBoT with your TI calculator, because once your battery gets low on the charge, you'll want to re-flash the factory firmware back on the CEENBoT so that you can re-charge your battery.

At the time this document was being written, the latest [factory] firmware can be obtained here:

`http://www.ceenbotinc.com/tools/`

You need to understand that if you wish to restore your CEENBoT to factory settings – for example, you want to use the CEENBoT to charge your battery after you've done experimenting with the TI driver software for the CEENBoT – that you need to re-*flash* your CEENBoT with the factory *firmware* (i.e., the aforementioned HEX file). It is assumed that the end-user understands how to perform this procedure. It is NOT within the scope of this document to discuss how this is done.

With that said, the following warnings should be taken seriously.

WARNING

Keep a backup of your original *firmware* and know how to *re-flash* your CEENBoT BEFORE YOU BEGIN EXPERIMENTING CEENBoT/TI DRIVER FIRMWARE!

Finally, and *most* importantly:

Note:

“TI”, and **“TI-Calculator”** are the trademarks of *Texas Instruments*.

Chapter 1: Introduction

This section introduces you to the CEENBoT/TI. It also covers how to flash the TI-driver firmware onto your CEENBoT for experimenting with the TI features. It also discusses the `Send()`/`Get()` commands on your calculator, which represent the foundation that makes the communication between the CEENBoT and the TI calculator possible.

Introduction

The CEENBoT can be flashed with what will henceforth be referred to as the *TI-driver firmware*, which is available as an Intel HEX file. This *firmware* gives you the ability to use various *Texas Instruments* calculator models to write programs on your TI calculator that can be used to control the CEENBoT. The TI-driver simply sits there waiting for incoming commands from the calculator, and once received, it determines what command it is, and the appropriate action can be taken.

Presently the software is still in its 'infancy' , but there are sufficient features that will allow potential users to 'test out' the capabilities of this new way of exploring the CEENBoT. This is possible by taking advantage of the capabilities warranted by the CEENBoT-API (Application Programming Interface) which presently dictates everything that the CEENBoT can and cannot do. The cool thing about the CEENBoT/TI integration is that this provides yet *another* means of exploring the CEENBoT, and when you write calculator programs on your TI, you are *essentially* writing programs that in the end, make use of the CEENBoT-API. Therefore, *you* are still writing CEENBoT-API programs. Refer to the *CEENBoT-API: Getting Started* guide and the *CEENBoT-API: Programmer's Reference* if you're curious about the CEENBoT-API.

Required Ingredients

The figure below pretty much summarizes what is needed for you to start exploring the CEENBoT with your TI calculator:

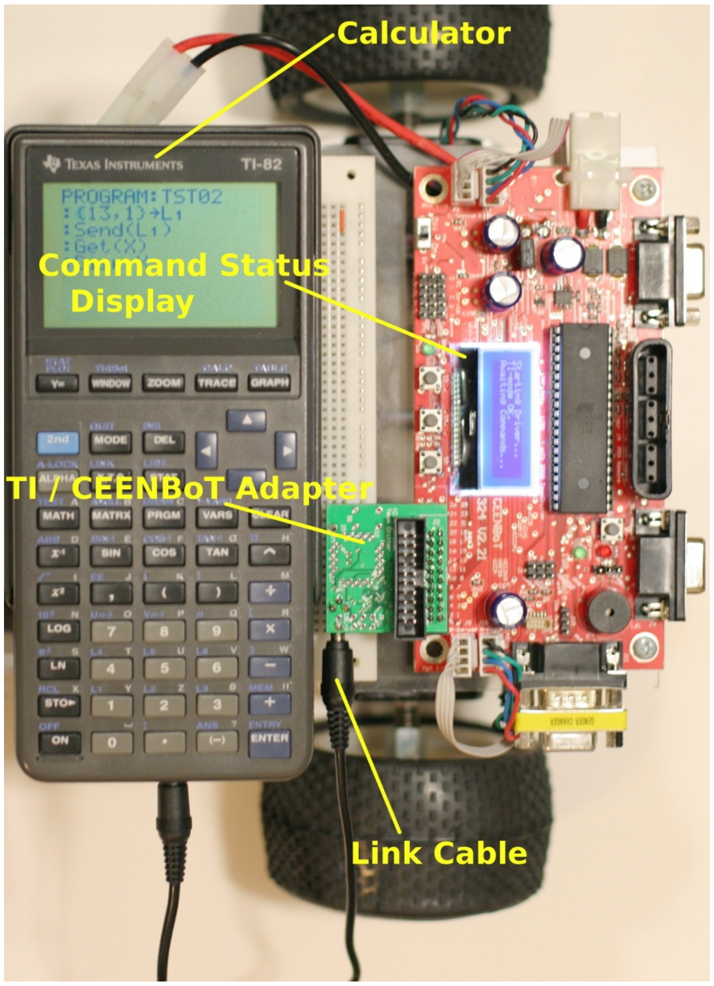
- **TI-Calculator** – supported (and *tested*) models are the `TI-82`, `TI-83+`, `TI-84+`, `TI-85/86`, `TI-89` (and variants).

- **CEENBoT –** It *must* be platform `'324 v2.21` or `v2.23`.

- **TI/CEENBoT Adapter Board (`v2.21`)** – This 'circuit-board' plugs onto the CEENBoT's connector (see figure). It is where the *link cable* connects to. Note the <u>new</u> `v2.23` boards <u>do NOT require that you have this adapter board</u> as the cable now connects directly *to* the CEENBoT controller board (*not shown*).

- **TI-communication Link Cable** – this is the communications cable with two 2.5mm jacks (similar to those used with your *headphones*, but the *jack* is smaller).

(*Continued on next page*)



- **AVR/ISP Programmer** – You need this to *flash* the TI-driver firmware if your CEENBoT doesn't come preloaded with it.

  Presently, since the TI-driver firmware is available for users to 'test run' the firmware on their own, you'll have to do the flashing of the firmware yourself. You'll need this (or a similar ISP programmer) in order to do so.

  This also means you'll need to have installed, or have access to ***AVR Studio*** from ATMEL Corporation, which is the software that can be used to upload and program the CEENBoT. This software is available for *free* from the ATMEL website at http://www.atmel.com (you will have to *search* for it and also *register* with ATMEL before you are allowed to download it).

- **TI-Driver Firmware** – This is the program that is flashed in the CEENBoT and allows communication between the CEENBoT and the TI calculator to take place. Presently, you can obtain trial versions from:

  `http://ceenbot.digital-brain.info`

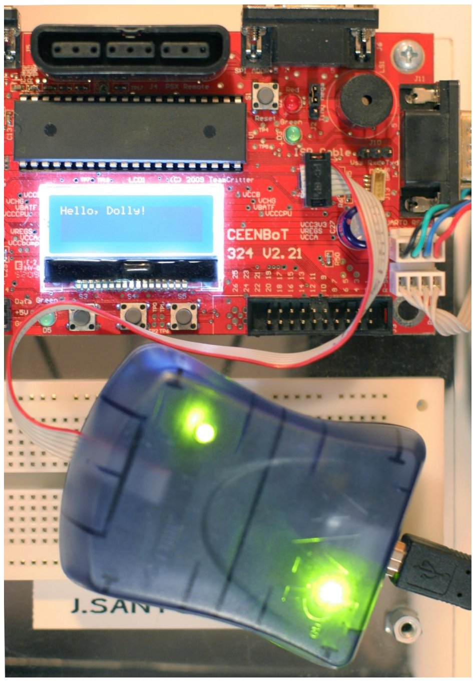
  Note: Manual programming of the TI-driver firmware into your CEENBoT is needed during this 'public testing' phase. However, in the future, it is expected that this feature will be part of the preloaded 'factory' firmware.

  Finally, please take notice also of the *Command Status Display* – this is the CEENBoT's LCD and it will display commands once received from the TI calculator.

Flashing the TI-Driver Firmware to your CEENBoT

Assuming that you have ***AVR Studio 4*** installed in your PC, and that you have a suitable ISP programmer (already *attached* to your PC's USB or COM port, depending on your ISP model) – you'd perform the following steps:

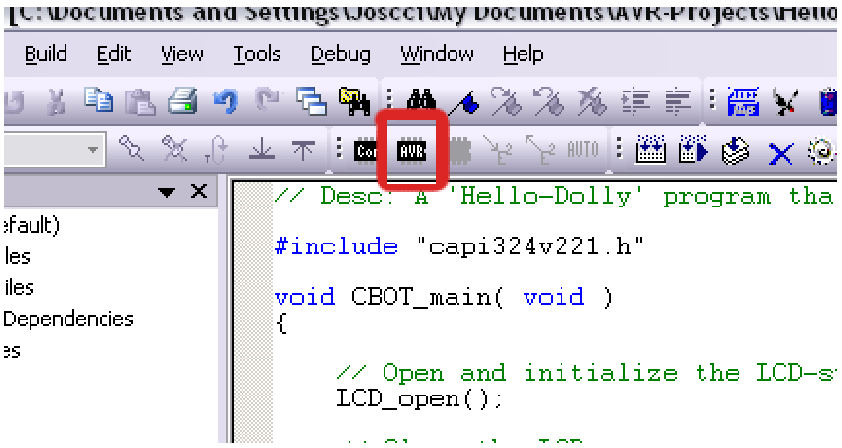
- Start *AVR Studio 4*.
- If the **`Welcome to AVR Studio`** panel shows up, simply click on **`[ Cancel ]`** since we're NOT creating a *project*.
- Connect your programer to your CEENBoT as shown below:



After attachment, TURN ON your CEENBoT.

Note: Make sure your calculator and TI-adapter are NOT attached to your CEENBoT.

- In *AVR Studio 4* there's an *icon* that says "AVR" as shown below – *click on it.*


[C:\Documents and Settings\Joscci\My Documents\AVR-Projects\Hello
Build Edit View Tools Debug Window Help
AVR
efault)
les
iles
Dependencies
es

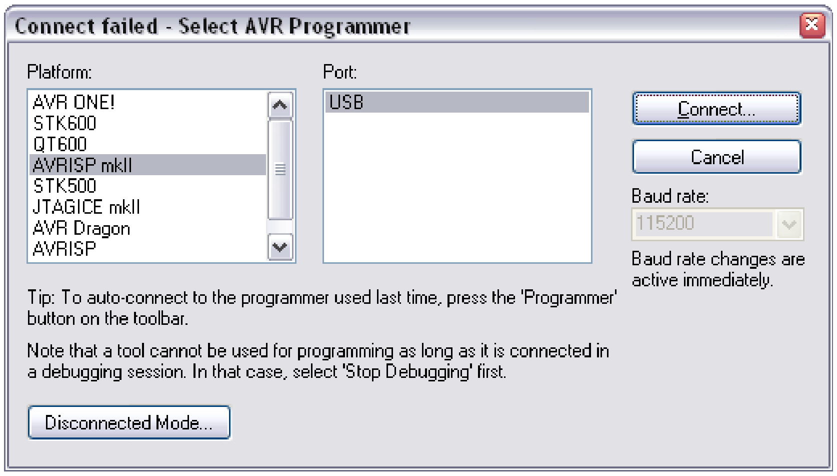
```
// Desc: A 'Hello-Dolly' program tha

#include "capi324v221.h"

void CBOT_main( void )
{

    // Open and initialize the LCD-s
    LCD_open();
```

- IF you're connecting your ISP for the *first time* the following panel might come up:



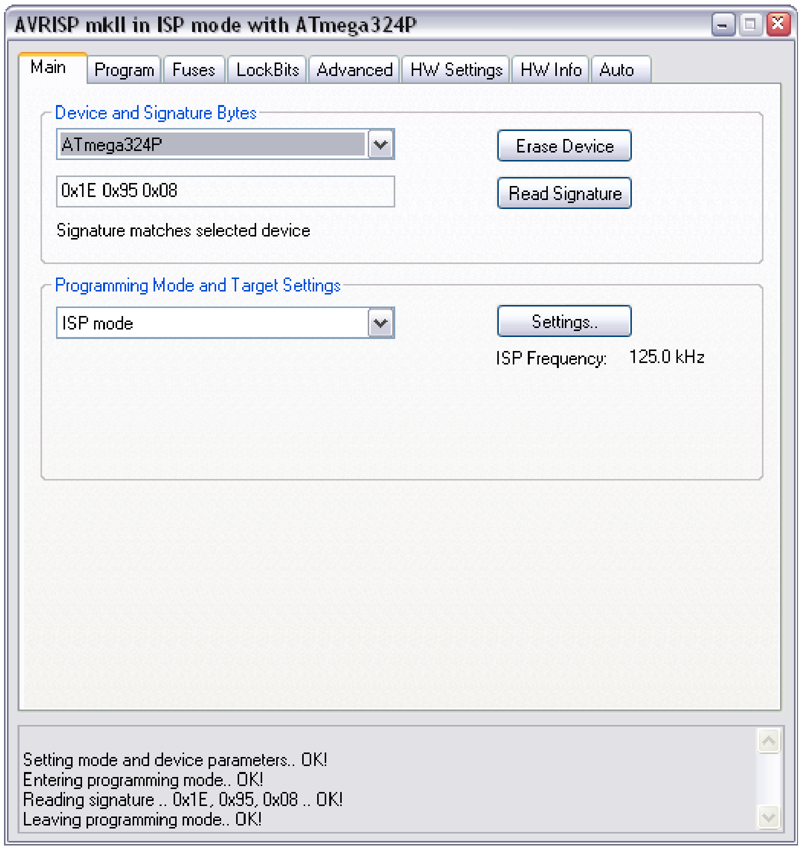
Baud rate changes are active immediately.

Tip: To auto-connect to the programmer used last time, press the 'Programmer' button on the toolbar.

Note that a tool cannot be used for programming as long as it is connected in a debugging session. In that case, select 'Stop Debugging' first.

Here, you must select the *model* (or Platform) and *Port* which your ISP programmer is attached to. On *my* machine, I'm using the **`AVRISP mkII`** on a **`USB`** port, so that's why I have that selected. Yours may be different. In any case, after selection, click on **`[ Connect ]`**.

- After clicking on the `[ Connect ]` button, the following *panel* should show up. Mine shows up with the **`Program`** Tab already selected, but yours might be different. In any case, before we do anything, click on the **`Main`** Tab first (you should see the following).


AVRISP mkII in ISP mode with ATmega324P
Main
Program
Fuses
LockBits
Advanced
HW Settings
HW Info
Auto
Device and Signature Bytes
Erase Device
Read Signature
Programming Mode and Target Settings
Settings..

ATmega324P

0x1E 0x95 0x08

Signature matches selected device

ISP mode

ISP Frequency: 125.0 kHz

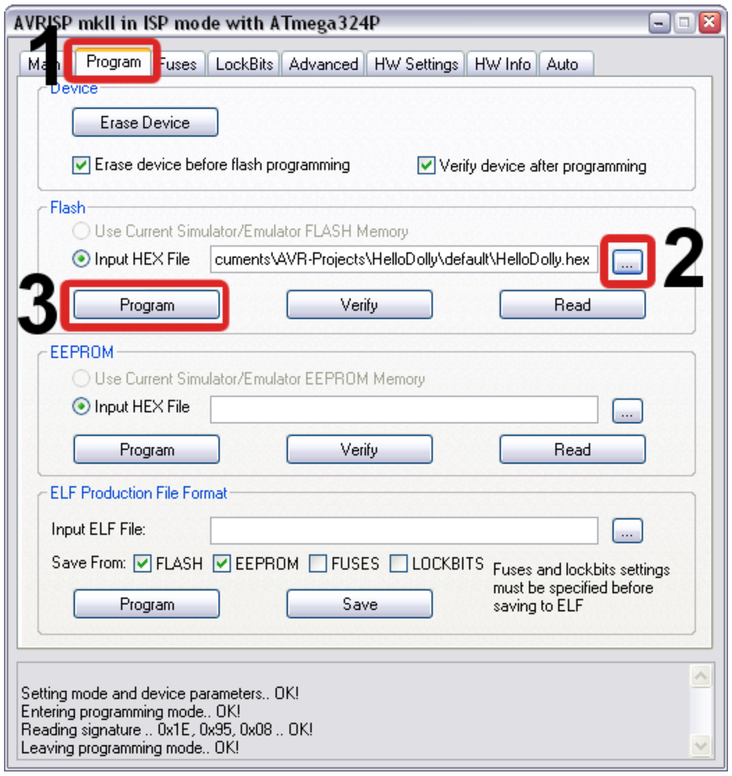
```
Setting mode and device parameters.. OK!
Entering programming mode.. OK!
Reading signature .. 0x1E, 0x95, 0x08 .. OK!
Leaving programming mode.. OK!
```

In the region labeled **`Device and Signature Bytes`**, click on the *drop-down* menu and select **`ATmega324P`**. This is the micro-controller on your **`CEENBoT '324 v2.2x`** platform.

Then, click on the **`[ Read Signature ]`** button. You should see messages along the bottom of the panel saying everything went "OK". If not you'll get an error.

Note: If this step is NOT successful then it is NOT worth trying to flash (program) your CEENBoT, because <u>this is a way for you to verify that your ISP programmer is successfully communicating with the micro-controller</u>. Therefore, <u>you CANNOT MOVE ON UNLESS YOU CAN SUCCESSFULLY READ THE SIGNATURE BYTES!</u>

- Now click on the **`Program`** Tab. You should see the panel below:


AVRISP mkII in ISP mode with ATmega324P
1
Main
Program
Fuses
LockBits
Advanced
HW Settings
HW Info
Auto
Device
Erase Device
Erase device before flash programming
Verify device after programming
Flash
Use Current Simulator/Emulator FLASH Memory
Input HEX File
2
3
Program
Verify
Read
EEPROM
Use Current Simulator/Emulator EEPROM Memory
Input HEX File
ELF Production File Format
Input ELF File:
Save From: FLASH EEPROM FUSES LOCKBITS
Save

[✓] Erase device before flash programming [✓] Verify device after programming

Input HEX File: cuments\AVR-Projects\HelloDolly\default\HelloDolly.hex

Save From: [✓] FLASH [✓] EEPROM [ ] FUSES [ ] LOCKBITS

Fuses and lockbits settings must be specified before saving to ELF

```
Setting mode and device parameters.. OK!
Entering programming mode.. OK!
Reading signature .. 0x1E, 0x95, 0x08 .. OK!
Leaving programming mode.. OK!
```

Click on **`[ ... ]`** as shown (2), and find the HEX file for the TI driver firmware (the above figure shows a different HEX file (I 'borrowed' the image above from a different *tutorial*)). You should know where you downloaded and saved the file on your system. Find it and select it (The TI-driver HEX file that is).

- Click on **`[ Program ]`**. Watch the progress bar go, and if everything goes well, no error messages will be issued.

  At this point, turn your CEENBoT OFF, disconnect the ISP table and proceed to the next section. You may now close *AVR Studio*.

Note: REMEMBER... the TI-driver firmware is provided for 'testing purposes'. It does NOT have power management, and it cannot monitor the voltage levels of your batteries. You must perform this same procedure to *flash* back the *original factory firmware* once you're done 'experimenting' with the TI-driver features if you want to charge your batteries with the CEENBoT and put everything back the way it was. <u>I cannot stress this enough</u>!

Starting the CEENBoT/TI Experience

Now that you have flashed the TI-driver firmware onto your CEENBoT, attach your calculator to the CEENBoT by using the *link* communication cable and plug that into the CEENBoT/TI adapter module (which should NOW be attached to your CEENBoT – if you have **`v2.21`** of the controller board), or directly into the jack of the controller board itself (if you have the new **`v2.23`** of the controller board).

First, turn the calculator ON, and *then* turn ON the CEENBoT.

Note: When you're done, do the above in the same order – turn the calculator OFF, *then* turn the CEENBoT OFF. If you turn the CEENBoT OFF first, then the calculator may become unresponsive and you won't be able to turn it OFF until you either unplug the link cable, and in some cases you *may* be forced to remove the batteries (if that happens to you).

It should display (note your firmware version *may* be different):

```
TI Driver v1.04.002R
(c) CEENBoT, Inc.
```

Then, about 2-4 seconds later it should display:

```
Starting Driver...
TI-mode OK.
Awaiting Commands...
```

If you see this – you're ready to rock and roll. However, if the CEENBoT has problems “seeing” the calculator it will display the following error message:

```
Starting Driver...
TI-mode ERROR.
```

Finally, before you start trying 'things' out , please take a look at the on-board liquid crystal display of the CEENBoT's controller board. As the CEENBoT sits there, awaiting for commands from the calculator, you should see an *angled bracket* (>) blinking ON and OFF at the top right-hand corner of the LCD at a rate roughly of 'once-per-second'. Think of this as the 'heartbeat' of the TI-driver program. As long as it's 'beating' – things are good... but if, while you're trying things out, you don't see this anymore, or it remains fixed (without blinking), then that tells you the firmware may have crashed or locked up. So just keep that in mind.

Let's write some programs!

First CEENBoT/TI Program

Let us begin with a super-simple example. We'll write a simple program to toggle the *red* LED on the CEENBoT's controller board.

Note: Before we begin, you should note that it is not within the scope of this document to teach you how to program using your calculator with TI-Basic – it is assumed that you know how to create a file to start writing a program in your calculator and know how to run it and are aware of the particular 'peculiarities' of your TI calculator model. The goal here is to give a *generalized* example.

Here's the code you should write on your TI calculator:

```
:{ 11, 1, 2 } → L₁
:Send(L₁)
:Get(X)
:Disp X
```

Note: Do NOT enter the above commands in a "letter-by-letter" fashion. That is, do not press the **[ALPHA]** key and then enter **S-e-n-d**, etc to enter commands. Instead, you must [after reading your calculator owner's manual] find the Send/Get commands under the suitable menu. Unfortunately, the manner in which this is done is different depending on the calculator model. For example, on the TI-83+/TI-84+, you push the **[PRGM]** button while already programming (editing your program), to get access to the command "catalog" supported by the calculator. If you scroll towards the bottom under the I/O menu, you'll find the Send/Get commands, which you select by pushing **[ENTER]**. But again, this is how it's done with the TI-83+/TI-84+ models. You have to consult the "programming" section of the manual of your TI calculator model so that you know how commands are entered for *your* particular TI model.

Let's talk about this program since it sets the tone for all the programs you will write. The first three lines are the *most* important because they represent the foundation for successful communication between the CEENBoT and the TI calculator – that is: the `Send()`, and `Get()` commands.

The first line creates a *set* (also called a *list*) containing three elements with the values given. The set is then *stored* in the variable L_1 (the arrow is obtained by pushing the **[STO>]** key on the calculator.

Note: Older TI modules require that lists be saved on *specific* variables such as L_1, L_2, L_3, etc. Newer calculator modules don't have this restriction, however. In that case you can 'store' the list on *any* variable, such as **X**, **Y** or **Z**.

Next, the `Send(L₁)` command works just as its name suggests – it *sends* the list of numbers via its communication port to the CEENBoT. The CEENBoT will read this list and determine its significance and perform some action as a result. In this case, the meaning is as follows:

11 – means this is an LED command.
1 – means this LED command will affect the *red* LED.
2 – means the LED will *toggle* states (if ON, then switch to OFF and vice versa).

Note: *Chapter 2* , consists of the supported *command reference*. You can check this chapter out to see what commands are supported, and the meaning behind the numbers of the list given.

Next is the `Get(X)` command. This will force the calculator to 'listen' for a response from the CEENBoT. The CEENBoT will respond by returning a value – this value will be stored in the named variable – in this case, it is `X`. Whether this value is of any significance depends on the *command* but the CEENBoT will always *return* a value upon issuing of a `Send()` from the calculator. Regardless of its significance you MUST always follow a `Send()` with a `Get()`!

Note: Every `Send()` command in your calculator program *must* be followed by a `Get()`. This means you should never issue multiple `Send()` command back-to-back, since there is no guarantee that the CEENBoT will catch all of them in this manner – the `Get()` ensures that the calculator waits for a response from the CEENBoT before proceeding with program execution.

Note: Do NOT use a *list variable name* – for example, **L₁**, with the `Get()` function. Use a generic variable, like **X**, **Y**, or **Z**. The calculator doesn't return *lists* back – it only receives them, so it doesn't make sense to store the result in a *list variable*.

Finally, the *last* line is optional. It is there to *show* on the calculator screen what was the value returned by the CEENBoT. This step is *not* necessary.

So as you can see, *all* of your CEENBoT/TI programs will consist of a series of `Send()/Get()` commands in addition to any other commands supported by your TI calculator. Please consult with the calculator's manual on writing programs for your calculator. In any case, and depending on your calculator model, you should be able to press the **[PRGM]** button to select your newly written program and execute it. If all goes well, you should see the angled bracket (>) on the LCD toggle ON and OFF each time the program is run.

Note: Newer calculators such as the TI-89 series don't have a **[PRGM]** button – program can be executed by entering your program's name with parenthesis – for example, if we had named our program **ledtest**. Then on the *home* screen, you would type `ledtest()` followed by the **[Enter]** key to begin execution. Again, all TI calculators have slight differences, so you must consult the manual for your particular model.

So, What Next?

Now that you've got the 'gist' of it, go to *Chapter 2* and familiarize yourself with the *command set* and start experimenting! I leave you with one final example program you can try out:

This is an example on how you can write a program on the TI-83+/TI-84+ models. Note that I don't make use of the *list variable* at all, and instead, write the list inside of the `Send()` function. I do this, because I can get away with doing that on *these* models – but that's not always the case. I labeled each line with a number for the brief discussion that follows – I encourage you to check out the command reference in the next chapter to understand how to 'decipher' each of the numerical values for each command:

```
: Send( { 12, 2, 300 } )            (1) – Sets the acceleration.
: Get( X )
: Send( { 1, 2, 0, 200 } )          (2) – Move in free-running mode.
: Get( X )
: Send( { 10, 4000 } )              (3) – Delay for 4 seconds.
: Get( X )
: Disp X
: Send( { 0, 2, 0 } )               (4) – Stop.
: Get( X )
: Send( { 3, 0, 150, 200, 1 } )     (5) – Tank-turn left for 150 steps.
: Get( X )
: Send( { 3, 1, 150, 200, 1 } )     (6) – Tank-turn right for 150 steps.
: Get( X )
```

(1) – Command **12**, sets the acceleration for both motors to **300 steps/sec**^{2}. The 2 indicates this command settings should affect *both* motors. This value is persistent. Once set, it stays set until you change it by issuing another command like it.

(2) – Command **1**, issues a motion in *free running mode*. In this mode, 'distance' is irrelevant. The motor just starts and it will only stop until you tell it to (see *next* command below). In this command, the **2** indicates both motors will move, while the **0** means to move *forward*. Finally, the **200** refers to the *speed* at which the motors will move (note that this is in addition to the *acceleration* set in step (1)).

(3) – Command **10**, is a *delay* command. The delay is in units of *milliseconds*. Therefore, the command as written in this line will delay for **4000ms**, which is equivalent to four *seconds*.

(4) – In this line, (which happens *four* seconds later), we issue command **0** to *stop* the motors that we started previously in line (2). The **2** that follows indicates that we want to stop *both* motors, and the **0** that follows indicates we wish to keep the 'brakes' *disengaged*.

(5) – (See *next* line below)
(6) – Lines (5) and (6) do the same thing. Command 3 is for “tank turn”. They issue a motion in *step mode*. In this mode, distance DOES matter, because we know specify the number of 'steps' each motor will move. Note the only difference between these two lines is the 0 (in line (5)) which causes the CEENBoT to “tank-turn” LEFT, and 1 (in line (6)) which causes the CEENBoT to “tank-turn” RIGHT. The **150** is the *number of steps* that the motors are to move, while the **200** is the *speed* that each motor will move, which in turn, determines how *fast* the CEENBoT “tank-turns”.

The above example focuses specifically on showing how the 'commands' are used via Send/Get. Keep in mind, that your programs can be more complicated than this – that is, you can use TI-Basic programming constructs to their full potential – you can use *while* loops, *for* loops, conditional *if..then...else* blocks, etc. It is up to you how creative you wish to get with your programming.

Chapter 2: Command Set Reference

This chapter covers the *command set* supported by the CEENBoT/TI-driver firmware. It covers the command set along with specific detail and meaning behind any required parameters that each command may or may not have.

Command Set Format

As discussed in Chapter 1, you send *commands* to the CEENBoT by constructing *lists*, which have one or more elements. The *first* element always designates **the command** that will be issued to the CEENBoT. Any additional elements that follow are additional arguments needed by the specific command to perform some action.

```
{ <command>, ... }
```

Throughout this chapter, descriptive labels will be used that are enclosed in angled brackets <> to give an idea of the meaning behind each *argument* that is part of the list. For example:

```
{ 11,  <which_LED>, <state> }
```

This *command* (`11`) can be used to control the LEDs on the CEENBoT. The first argument *must* be 11, while the meaning of the *second* and *third* arguments are used to specify *which* LED is affected, and what the operation or *state* of the LED will take place. However, keep in mind that you can only specify *integers* for all *arguments* of the command.

Command Set Summary

The following is a summary of the *command* set supported by the CEENBoT/TI firmware.

Motion Commands

- `{0, ... }` – Stop any current motion.
- `{1, ... }` – Issue a motion command in *free-running* mode.
- `{2, ... }` – Issue a motion command in *step* mode.
- `{3, ... }` – Issue a motion command to do a *tank turn* (in *step* mode).
- `{4, ... }` – Issue a motion command in *free-running* mode and STOP if you bump into something.
- `{5, ... }` – Issue a motion command in *step* mode and STOP if you bump into something.

Utility Commands

- `{10, ... }` – Delay for a specified number of *milliseconds*.
- `{11, ... }` – Used to control the state of the on-board LEDs.
- `{12, ... }` – Used to set the *acceleration* of the stepper motors (in **`steps/sec`**^{2}).
- `{13, ... }` – Used to get state information about the IR (Infra-Red) sensors, and on-board switches.
- `{14, ... }` – Used to set the *speed* of the stepper motors (in **`steps/sec`**). This command will also *issue* a motion.
- `{15, ... }` – Used to set the *direction* of the stepper motors when 'primitive' motion commands are issued (such as command 14 above, to set the speed).
- `{16, ... }` – Used to set the number of *steps* that each stepper should move when the operating run mode is *step* mode.
- `{17, ... }` – Used to set the operating *run* mode.

(*Continued on next page*)

- `{18, ... }` – Used to control the positions of any RC servos attached to the CEENBoT.
- `{19, ... }` – Used to play a repetitive 'beep pattern' via the CEENBoT's speakers.
- `{20, ... }` – Used to trigger the ultrasonic sensor and obtain a distance-to-target.

Event Commands

(7) `{30, ... }` – The *wait-on-bump* command. Used to pause execution of your program until one of the IR bump sensors is triggered.
(8) `{31, ... }` – The *wait-on-switch* command. Used to pause execution of your program until one of the three switches is depressed.

Test Command

- `{2000, ... }` – This is a *test* command (i.e., for *testing* purposes) – it does nothing.

Command Set Reference

The `STOP` Command: `{0, ... }`

Format:

```
{ 0, <which>, <brake_mode> }
```

Description:

Use this command to *stop* or *cancel* a motion that is currently taking place.

Arguments:

`<which>` – Must be one of the following values:

```
0 = LEFT stepper.
1 = RIGHT stepper.
2 = BOTH steppers.
```

`<brake_mode>` - Must be one of the following values:

```
0 = Keep the brakes OFF upon stopping.
1 = Engage the stepper brakes.
```

> **Note:** If you engage the brakes, they will remain engaged until you explicitly dis-engage them! This can be done by re-issuing *this* same command, but with the *brake mode* argument set to 0.

The `RUN` Command: `{1, ... }`

Format:

```
{ 1, <which>, <dir>, <speed> }
```

Description:

This command can be used to issue a motion (get the stepper motors moving) in *free-running* mode. In this mode, the motors will run for an indefinite amount of time (until you issue a **`STOP`** command).

Arguments:

`<which>` – Must be one of the following values – it determines which motor will be affected:

```
0 = LEFT stepper.
1 = RIGHT stepper.
2 = BOTH steppers.
```

`<dir>` – Must be one of the following values:

```
0 = Move specified motor(s) forward.
1 = Move specified motor(s) in reverse.
```

(*Continued on next page*)

(*Continued from previous page*)

`<speed>` - Specifies the speed of the affected motor(s) in *steps/sec*.

Must be between `0` and `400`.

The `STEP` (& Wait) Command: `{ 2, ... }`

Format:

```
{ 2,  <which>, <dir>, <steps>, <speed>, <brk_mode> }
```

Description:

This command can also be used to issue a motion (get the stepper motors to move), but is for issuing *finite-distance* moves. That is, a motion in *step* mode (as opposed to *free-running* mode). In this mode, you specify precisely how far you want to move by specifying the number of 'steps' the affected stepper(s) will move. Therefore, motion is bound to complete at some point. The function will BLOCK (*wait*) until the motion has fully completed.

Arguments:

`<which>` – Must be one of the following values – it determines which motor will be affected:

```
0 = LEFT stepper.
1 = RIGHT stepper.
2 = BOTH steppers.
```

`<dir>` – Must be one of the following values:

```
0 = Move specified motor(s) forward.
1 = Move specified motor(s) in reverse.
```

`<steps>` - Specifies the 'distance' (how many steps the affected stepper(s) will move).

Must be a value between `0` and `65535` (approximately `327` wheel revolutions).

`<speed>` - Specifies the speed of the affected motor(s) in *steps/sec*.

Must be between `0` and `400`.

`<brake_mode>` - Must be one of the following values:

```
0 = Keep the brakes OFF upon stopping.
1 = Engage the stepper brakes.
```

Note: If you engage the brakes, they will remain engaged until you explicitly dis-engage them! This can be done by re-issuing *this* same command, but with the *brake mode* argument set to 0.

The `TANK-TURN` Command: `{ 3, ... }`

Format:

```
{ 3, <which_way>, <steps>, <speed>, <brk_mode> }
```

Description:

This command like **`STEP & WAIT`** previously discussed can also be used to issue finite-distance moves in *step* mode. However, this command is specifically for *turning* the CEENBoT in a tank-like manner (about the axis located at the center between the CEENBoT's two front wheels.

Arguments:

`<which_way>` – Must be one of the following values:

`0 = Turn LEFT.`
`1 = Turn RIGHT.`

`<steps>` - Specifies the 'distance' (how many steps the affected stepper(s) will move).

Must be a value between **`0`** and **`65535`** (approximately **`327`** wheel revolutions).

`<speed>` - Specifies the speed of the affected motor(s) in *steps/sec*.

Must be between **`0`** and **`400`**.

`<brake_mode>` - Must be one of the following values:

`0 = Keep the brakes OFF upon stopping.`
`1 = Engage the stepper brakes.`

Note: If you engage the brakes, they will remain engaged until you explicitly dis-engage them! This can be done by re-issuing *this* same command, but with the *brake mode* argument set to 0.

The `RUN-AND-BUMP` Command: `{ 4, ... }`

Format:

```
{ 4, <which>, <dir>, <speed>, <timeout> }
```

Description:

This command will issue a motion in *free-running* mode, but it will WAIT UNTIL either of the following happens:

- LEFT and/or RIGHT IR bump sensors are triggered (due to obstacle).
- The specified *timeout* elapses completely.

The purpose of the timeout is for the CEENBoT to run forever if no object is ever encountered, which will also keep your calculator program from running if a timeout doesn't happen later or sooner and is the reason for its existence.

Arguments:

`<which>` – Must be one of the following values – it determines which motor will be affected:

`0 = LEFT stepper.`
`1 = RIGHT stepper.`
`2 = BOTH steppers.`

`<dir>` – Must be one of the following values:

`0 = Move specified motor(s) forward.`
`1 = Move specified motor(s) in reverse.`

`<speed>` - Specifies the speed of the affected motor(s) in *steps/sec*.

`Must be between 0 and 400.`

`<timeout>` – Must be some non-zero value up to 30 seconds (1 to 30). It specifies how long the CEENBoT should continue to move in the specified direction until it has to stop if no object collision ever occurs.

Returns:

The CEENBoT will return the following values upon completion of this command, which is obtained by the user by issuing a `Get()` following the corresponding `Send()` on the calculator program – the returned value represents the *reason* for having stopped:

`0 = No bump hit (i.e., timed out).`
`1 = LEFT bump triggered.`
`2 = RIGHT bump triggered.`
`3 = BOTH bump sensors triggered simultaneously.`

The user then can use this information to make further decisions in his/her program (i.e., move around the offending obstacle, back up, etc).

The `STEP-AND-BUMP` Command: `{ 5, ... }`

Format:

```
{ 5, <which>, <dir>, <speed>, <steps> }
```

Description:

This command will issue a motion for a *finite-distance* move in *step-mode*. It will continue this motion until one of the following occur:

- A *bump* sensor is triggered (LEFT, RIGHT, or BOTH) due to obstacle detection.
- The initial specified *step distance* to travel has been completed.

In either case the 'BoT will come to a stop and will return back a value that signifies the reason for stopping (see 'Returns' section below).

Arguments:

`<which>` – Must be one of the following values – it determines which motor will be affected:

```
0 = LEFT stepper.
1 = RIGHT stepper.
2 = BOTH steppers.
```

`<dir>` – Must be one of the following values:

```
0 = Move specified motor(s) forward.
1 = Move specified motor(s) in reverse.
```

`<speed>` - Specifies the speed of the affected motor(s) in *steps/sec*.

Must be between 0 and 400.

`<steps>` - Specifies the 'distance' (how many steps the affected stepper(s) will move).

Must be a value between **0** and **65535** (approximately 327 wheel revolutions).

Returns:

The CEENBoT will return the following values upon completion of this command, which is obtained by the user by issuing a `Get()` following the corresponding `Send()` on the calculator program – the returned value represents the *reason* for having stopped:

```
0 = No bump hit while traveling.
1 = LEFT bump triggered.
2 = RIGHT bump triggered.
3 = BOTH bump sensors triggered simultaneously.
```

The user then can use this information to make further decisions in his/her program (i.e., move around the offending obstacle, back up, etc).

Utility Commands

The previous section discussed *motion-issuing* commands – in this section, we cover 'utility' commands (miscellaneous commands other than motion-issuing types).

The `DELAY` Command: `{ 10, ... }`

Format:

```
{ 10, <delay_ms> }
```

Description:

This command can be used to *issue* a delay in units of *milliseconds* to have the CEENBoT (and your Calculator program) *hold off* before doing anything else.

Arguments:

`<delay_ms>` – The delay to wait on in units of *milliseconds*.

Must be between **`0`** and **`32767`** *milliseconds*. (**`32.7`** seconds max).

The `LED` Command: `{ 11, ... }`

Format:

```
{ 11, <which_LED>,  <state> }
```

Description:

This command can be used to control the on-board LEDs on the CEENBoT (Red and Green LEDs). They can be turned ON, OFF, or *toggled* from one state to the other.

Arguments:

`<which_LED>` – Specifies *which* LED will be affected:

```
0 = Green LED.
1 = Red LED.
```

`<state>` – Specifies *how* the LED will be affected:

```
0 = Turn the LED OFF.
1 = Turn the LED ON.
2 = Toggle the LED STATE. (From ON to OFF and vice versa).
```

The `SET ACCELERATION` Command: `{ 12, ... }`

Format:

```
{ 12, <which>, <accel> }
```

Description:

This command can be used to enable or disable stepper *acceleration* by setting the appropriate acceleration constant. Acceleration is given in units of **`steps/sec`**^{2}.

Arguments:

`<which>` – Must be one of the following values – it determines which motor will be affected:

```
0 = LEFT stepper.
1 = RIGHT stepper.
2 = BOTH steppers.
```

<accel> - The acceleration rate for the specified stepper(s) in units of **`steps/sec`**^{2}:

Must be between `0` to `1000`.

Note: A value of zero acceleration effectively disables *acceleration*. Any *non-zero* value will enable the acceleration.

The `GET SENSORS` Command: `{ 13, ... }`

Format:

```
{ 13, <which_sensor> }
```

Description:

This command can be used to inquire about the state of the on-board Infrared (IR) sensors as well as the state of the on-board switches of the CEENBoT. A user can then use the returned value to make a decision in his/her Calculator program. As with all commands you *must* follow this command with a Get() call in order to obtain the 'response' for the information you requested from the CEENBoT regarding the state of the sensors.

Arguments:

`<which_sensor>` – Must be one of the following:

```
0 = Get state of LEFT IR (Bump sensor).
1 = Get state of RIGHT IR (Bump sensor).
2 = Get state of either LEFT and/or RIGHT (whichever is active if any).
3 = Get state of SW3 (on-board switch).
4 = Get state of SW4 (on-board switch).
5 = Get state of SW5 (on-board switch).
```

(*Continued on next page*)

(*Continued from previous page*)

Returns:

The command will return either a 1 or a 0 regarding the state of the *requested* sensor. To obtain this value, you must follow the `Send()` command with a `Get()` to obtain the response from the CEENBoT. For example:

```
:{ 13, 2 } → L₁
:Send(L₁)
:Get( X )
:Disp X
```

Request the state of *either* IR sensor. Then, the following `Get()` will obtain the result. This value will be **`1`** if either IR sensor is being 'blocked', or **`0`** if nothing is happening.

The SET RC SERVO POSITION Command: `{ 18, ... }`

Format:

```
{ 18, <which_servo>, <position> }
```

Description:

This command allows an RC (Radio-controlled) servo to be controlled – that is, for its position to be manipulated. An RC servo can be attached to the CEENBoT via one of the *five* RC servo ports **`0`**, **`1`**, **`2`**, **`3`**, or **`4`** (located next to the ON/OFF switch).

Arguments:

`<which_servo>` – Must be one of the following:

```
0 = Position value will affect the servo on port 0.
1 = Position value will affect the servo on port 1.
2 = Position value will affect the servo on port 2.
3 = Position value will affect the servo on port 3.
4 = Position value will affect the servo on port 4.
```

`<position>` – Must be a value between **`400`** and **`2100`**. These values correspond to the counter-clockwise and clockwise-most positions of the servos. Finding the necessary numerical value such that your servo sits at its positional 'center' must be determined experimentally by you.

The PLAY BEEP PATTERN Command: `{ 19, ... }`

Format:

```
{ 19, <beep_freq>, <beep_duration_ms>, <beep_active_percent>, <beep_repeat_times> }
```

Description:

This command allows you to generate some interesting (although not necessarily *musical*) beep patterns via the CEENBoT's on-board speaker. Beeps can be emitted up to a frequency of 500Hz, so it is by no means for musical purposes, but beeping, for "beeping's sake".

Arguments:

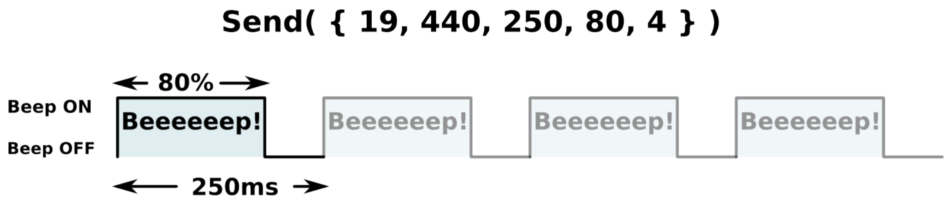
`<beep_freq>`	– This is the *beep frequency* in *Hz*. It must be a value between **`0`** to **`500`**.
`<beep_duration_ms>`	– This argument specifies the duration (in *milliseconds*) of the beep. It must be a value between **`0`** to **`32767`** ms (equivalent to 32.767 seconds).
`<beep_active_percent>`	– This must be a percent value between **`0`** and **`100`**. This is the percentage of time of the *beep duration* (given in the previous argument) for which the note will remain active (i.e., *audible*). For example, if `<beep_duration_ms>` above is **`250`** ms, and `<beep_active_percent>` is **`80`**, then of the **`250`** ms, the note will be audible for **`200`** ms (that's 80% of 250), and silent for the remaining **`50`** ms. But it will still take 250ms for the command to complete.
`<beep_repeat_times>`	– This is the number of times you want to repeat the beep. If you specify **`1`**, then the beep will play once. But if you specify **`4`**, or **`8`**, or *whatever*, then the note will play multiple times and thus, generate the *beep pattern*. Of course, this requires that you ensure your beep is *silent* for a little bit (by *not* specifying **`100`** for the `<beep_active_percent>` argument, so that you can hear the beep stop and start again – otherwise, your beep pattern will sound as one single *long* beep.

Example:

```
: Send( { 19, 440, 250, 80, 4 } )            (1)
: Get( X )
: Send( { 19, 500, 125, 80, 8 } )            (2)
: Get( X )
```

Line (1) we issue a beep pattern of 400Hz that takes 250ms to complete, but is only audible for 80% of *this* time. The note is repeated 4 times. (See the below figure as an illustrative example).

Line (2) we issue another beep pattern, this time at 500Hz that takes 125ms to complete, but is only audible for 80% of *this* time. The note is repeated 8 times.



Example of a 'single' beep pattern played four times back-to-back

The PING Command: `{ 20, ... }`

Format:

```
{ 20, <number-of-times-to-ping> }
```

Description:

This command allows you to *trigger* the **PARALLAX Ultrasonic PING)))** sensor to emit an "echo pulse" that will hit an object and bounce back, allowing you to determine the distance-to-target. The command returns the distance-to-target in centimeters scaled by **10**, meaning that you *must* divide the value returned by **10** to get the distance in pure centimeters. See the 'Example' section below.

Argument:

`<number-of-times-to-ping>` – This is the number of 'echoes' to emit, before returning the distance-to-target. The function will wait **10ms** before emitting the next echo. If more than one 'echo' is requested, the distance value returned will be the average of all the echoes specified. Thus, you must specify a value of 1 or more. (See 'Example' below).

Returns:

The CEENBoT will return the the distance-to-target in units of *centimeters* scaled by **10** upon completion of this command, which is obtained by the user by issuing a '`Get()`' following the corresponding '`Send()`' on the calculator program. Because the value is "scaled by 10", you must, on the calculator side, divide the returned value by **10**. This resulting value, is the distance-to-target in pure *centimeters*.

Hardware Dependencies:

To make use of this feature you'll need the **PARALLAX Ultrasonic PING)))** sensor presently used with the CEENBoT. Aside from the *Power* and *Ground* pins to power the sensor, the 'trigger' pin must be connected to **PA3**, accessible via header **J3**, pin **1** on the CEENBoT's controller board.

Power and Ground to power the sensor can be obtained from the pins in header **J7** where the RC servos plug into. You can use an *un-used* port to power your sensor. Detailed instructions are beyond the scope of this document.

Example:

Here's a quick example of a TI-Basic program to obtain distance-to-target using the ultrasonic sensor. The numbers in parenthesis are simply for reference and are not part of the program:

```
:Send( {20, 4} )              (1)
:Get(X)               (2)
:(X/10)->Y            (3)
:Disp Y               (4)
```

Line **(1)** sends the command to issue an 'echo' *four* times. Line **(2)** obtains the result and stores the distance value in the variable `X`. Then in line **(3)**, `X` is divided by `10` and stored in `Y` where in line **(4)** we display the result (which is in units of *centimeters*). You can use other conversion values in your program to convert from centimeters to some other distance units.

Event Commands

Event commands are commands that *wait* for something to “happen”. Event commands cause your program to “pause” until the requested event occurs. Currently two event commands exist.

The `WAIT-ON-BUMP` Command: `{ 30, ... }`

Format:

```
{ 30, <which_sensor> }
```

Description:

The *wait-on-bump* command will cause your program to “pause” until the specified IR bump sensor is triggered. You can specify whether you want to “wait” until just the LEFT sensor is triggered, or just the RIGHT sensor, or *either* sensor.

Arguments:

`<which_sensor>` – Must be one of the following values:

```
1 = Wait until the LEFT bump sensor is triggered.
2 = Wait until the RIGHT bump sensor is triggered.
3 = Wait until either the LEFT or RIGHT (or BOTH) sensors are triggered.
```

Returns:

The command will return one of the following values, which you obtained by following the `Send()` command with the corresponding `Get()`:

```
1 = The LEFT bump sensor was triggered.
2 = The RIGHT bump sensor was triggered.
3 = BOTH bump sensors were simultaneously triggered.
```

Note that you will only be notified to the sensor you specified in `<which_sensor>`. For example, if you specify that you want to wait until the LEFT sensor is triggered, then you will only be notified if the LEFT sensor is triggered and not on the RIGHT, etc.

The `WAIT-ON-SWITCH` Command: `{ 31, ... }`

Format:

```
{ 31, <which_switch> }
```

Description:

This *wait-on-switch* command will cause your program to “pause” until the specified switch is depressed. You can specify whether you want to wait until switch **`3`**, or **`4`**, or **`5`** are depressed. These are labeled **`SW3`**, **`SW4`**, and **`SW5`** on the CEENBoT's controller board. Note that *nothing* else will happen until the condition you're waiting on occurs.

Arguments:

`<which_switch>` – Must be one of the following values, which specifies the *switch* you're waiting on:

```
0 = Wait until ANY switch is depressed.
3 = Wait until switch SW3 is depressed.
4 = Wait until switch SW4 is depressed.
5 = Wait until switch SW5 is depressed.
```

Returns:

The command returns a numerical value, which you obtain by issuing a `Get()` after the corresponding `Send()` signifying the switch that was depressed that caused the “wait” to terminate:

```
3 = SW3 was depressed.
4 = SW4 was depressed.
5 = SW5 was depressed.
```

Test Commands

The `TEST` Command: `{ 2000, ... }`

Format:

```
{ 2000, <val_1>, <val_2>, <val_3>, <val_4>, <val_5> }
```

Description:

This command can be used to 'test' communication between the attached calculator and the CEENBoT. It doesn't do anything other than it displays the received values and displays them on the LCD display of the CEENBoT. It is for debugging purposes, but you can use it to check and verify communication between your calculator and the CEENBoT.

Arguments:

`<val_1>` – Can be any value in the range of **`-32768`** to **`32767`**.
`<val_2>` – Can be any value in the range of **`-32768`** to **`32767`**.
`<val_3>` – Can be any value in the range of **`-32768`** to **`32767`**.
`<val_4>` – Can be any value in the range of **`-32768`** to **`32767`**.
`<val_5>` – Can be any value in the range of **`-32768`** to **`32767`**.

Returns:

The command returns the value of **`2000`**, which can be obtained by issuing a `Get()` following the corresponding `Send()` that initiated the command in the first place.

Primitive Motion Commands

The following commands can be used to issue *motions* (get the stepper motors to move) – however this are slightly more *primitive* commands. The preferred motion commands should be those discussed at the beginning of this chapter: RUN, STEP (& WAIT), and TANK-TURN. However, the commands that follow can be used for 'finer control'. They're included here for 'whatever it's worth'. Unlike the principal motion commands at the beginning of this chapter, these commands require that you *explicitly* set the operating run mode before doing anything else – recall the operating run modes are *free-running* and *step* modes.

The `RUNMODE` Command: `{ 17, ... }`

Format:

```
{ 17, <run_mode> }
```

Description:

Use this command to set the operating run mode when using the primitive motion commands. The two operating modes are free-running in which (distance plays NO role), and step mode (where distance DOES play a role – i.e., finite-distance move).

Arguments:

`<run_mode>` – Must be one of the following:

0 = Set run mode to *free-running*.
1 = Set run mode to *step* mode.

The `SET DIRECTION` Command: `{ 15, ... }`

Format:

```
{ 15, <dir_L>, <dir_R> }
```

Description:

This command can be used to *prepare* the direction the wheels will move ONCE motion is issued with the **SET SPEED** *primitive* command.

Arguments:

`<dir_L>` – Must be one of the following:

0 = LEFT wheel should move forward.
1 = LEFT wheel should move in reverse.

`<dir_R>` – Must be one of the following:

0 = RIGHT wheel should move forward.
1 = RIGHT wheel should move in reverse.

The `SET STEPS` Command: `{ 16, ... }`

Format:

```
{ 16, <which>, <steps_L>, <steps_R> }
```

Description:

This command can be used to *prepare* the distance that each stepper will move ONCE motion is issued with the **`STEP SPEED`** *primitive* command. Specification of distance with this command is *required* if you have set the operating *run mode* to *step* mode using the **`RUNMODE`** command.

Arguments:

`<which>` – Must be one of the following values – it determines which motor will be affected:

```
0 = LEFT stepper.
1 = RIGHT stepper.
2 = BOTH steppers.
```

`<steps_L>` – Must be a value between **`0`** and **`65535`** (approximately **`327`** wheel revolutions).
`<steps_R>` – Must be a value between **`0`** and **`65535`** (approximately **`327`** wheel revolutions).

Note: Note that if you specify only ONE stepper to be affected, the parameter for the *other* will be ignored – (i.e., you can set it to zero). Regardless of whether you specify that only one motor, or BOTH motors are affected, *all* parameters are REQUIRED – even if you have to set one of them to zero – it is very likely this 'awkwardness' will change in a future revision.

The `SET SPEED` Command: `{ 14, ... }`

Format:

```
{ 14, <which>, <speed_L>, <speed_R> }
```

Description:

All of the *primitive motion commands* discussed so far simply 'prepare' the parameters for a motion to take place (the *run mode*, *steps* (if in *step* mode), and *direction*, etc). THIS command, then, *is* THE command which will execute and trigger the motion to take place according to the previously set parameters.

(*Continued on next page*)

(*Continued from previous page*)

Arguments:

`<which>` – Must be one of the following values – it determines which motor will be affected:

```
0 = LEFT stepper.
1 = RIGHT stepper.
2 = BOTH steppers.
```

`<speed_L>` – Must be a value between **0** and **400.**
`<speed_R>` – Must be a value between **0** and **400.**

Note: Note that if you specify only ONE stepper to be affected, the parameter for the *other* will be ignored – (i.e., you can set it to zero). Regardless of whether you specify that only one motor, or BOTH motors are affected, *all* parameters are REQUIRED – even if you have to set one of them to zero – it is very likely this 'awkwardness' will change in a future revision.

Note: The **`SET SPEED`** command can also be useful for *modulating* the speed of the steppers as a motion is TAKING PLACE (in *free-running* mode ONLY) – that is, a motion you may have already issued via **`RUN`** command discussed at the beginning of this chapter.

In this case it is *not* necessary to set the direction, operating mode, nor number of steps since the *higher* motion commands do all this stuff internally.