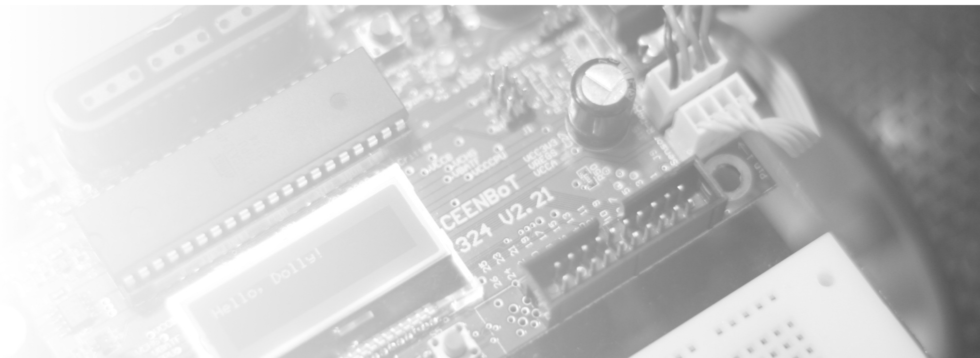
The CEENBoT-API: Programmer's Reference

Programmer's Reference Guide for the CEENBoT
CEENBoT v2.21 – '324 Platform



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

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

Rev 1.08
(Current as of: **v1.09.000R**)

Copyright © 2011, University of Nebraska Board of Regents

(Blank)

STOP!

Please make sure to read the *Getting Started* guide before referring to this document. The Getting Started guide contains important preliminary information it is assumed you know and understand prior-hand.

NOTE

This is a *preliminary document*. Some features may or may not be yet readily available and as development of the API 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 API libraries/header files.

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 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 firmware before you start writing programs with this API.

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 CEENBoT-API – that you need to r*e-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.

with a 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 WITH THE CEENBoT-API!

Finally, and *most* importantly:

> **NO WARRANTY**
>
> THIS PROGRAM ("THE CEENBOT API") or simply ("API") IS DISTRIBUTED IN THE HOPE THAT IT WILL BE USEFUL, BUT WITHOUT ANY WARRANTY. IT IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
>
> IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW THE AUTHOR WILL BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA AND/OR EQUIPMENT OR DATA/EQUPMENT BEING RENDERED INACCURATE OR USELESS OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM OR DEVICEE TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF THE AUTHOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.
>
> **YOUR USE OF THIS "API" CONSTITUTES YOUR AGREEMENT AND UNDERSTANDING OF THE 'NO-WARRANTY' CLAUSE.**

Table of Contents

Note: If you're viewing the PDF document the pages below *link* to the corresponding chapters.

- **Chapter 1** – Introduction to the API **pg. 9**
- **Chapter 2** – The CAPI subsystem module **pg. 17**
- **Chapter 3** – The ADC subsystem module **pg. 21**
- **Chapter 4** – The ISR subsystem module **pg. 27**
- **Chapter 5** – The LCD subsystem module **pg. 35**
- **Chapter 6** – The LED subsystem module **pg. 49**
- **Chapter 7** – The PSXC subsystem module **pg. 59**
- **Chapter 8** – The SPI subsystem module **pg. 75**
- **Chapter 9** – The SPKR subsystem module **pg. 85**
- **Chapter 10** – The STEP subsystem module **pg. 107**
- **Chapter 11** – The SWATCH subsystem module **pg. 143**
- **Chapter 12** – The TINY subsystem module **pg. 151**
- **Chapter 13** – The TMRSRVC subsystem module **pg. 163**

    Introduction to CRC functions pg. 177
- **Chapter 14** – The UART subsystem module **pg. 185**
- **Chapter 15** – The USONIC subsystem module **pg. 199**
- **Chapter 16** – The I2C subsystem module **pg. 205**
- **Chapter 17** – Useful HELPER utilities **pg. 231**

Chapter 1: Introduction to the CEENBoT-API

This chapter introduces you to the CEENBoT-API – what it is and how it is structured.

Introduction to the CEENBoT-API

The Basics

The CEENBoT-API is an *application programming interface* that exposes a set of functions which allow you to control and manipulate the CEENBoT in a simplified manner via well-documented function calls. The purpose is not necessarily to replace 'bare-metal' programming of the 'BoT, but simply to provide an optional, or more appropriately, an alternative approach that allows the user to explore the CEENBoT and its capabilities in a more friendly, inviting, and open-ended manner. Consequently, the goal is to 'open up' the CEENBoT for exploration at multiple skill levels for those who do not wish (for whatever reason) to bother with the intricate details of the CEENBoT's electronics.

The CEENBoT-API exposes a rich set of C functions that allows various hardware resources available on the CEENBoT itself to be easily manipulated. Some of these hardware resources can include the peripherals embedded on the MCU itself, such as control of I^2C (or TWI), SPI, or UART subsystems to name a few. This is in addition to additional on-board peripherals on the CEENBoT itself such as the ability to manipulate the on-board LCD display, LEDs, and *stepper* motors for mobility.

The entire set of API functions is exposed to the user by way of a *pre-compiled static library* targeting the supported platform, which at the time of this writing this constitutes the '324 v2.21 board. In essence, with the appropriate [supplied] *header files* along with the *static library* file, the user merely includes these header files and links against the static library to take advantage of the API. The process on how exactly this is accomplished is outlined in detail in the *Getting Started* guide, which should be accessible to you in the same place you obtained this document.

Modular Structure of the CEENBoT-API

The CEENBoT-API is organized into similarly related functions called *modules*. Each *module* is in charge of acquiring the necessary resources (such as memory, I/O port pins, or peripherals on the MCU) to either achieve a task, or control a particular peripheral device (such as the on-board LCD display). The following is a list of the existing *modules* available through the API and a brief description of its purpose or function. It will be the convention in this document to list the name of a supported module using capital letters. Sometimes the term '*subsystem module*' or '*subsystem/module'* is used to denote the same meaning – that is to mean a *module* that specifically supports a given *subsystem* that is either embedded on the MCU or as an external peripheral of some sort. Here's the list of ready-modules available for use. Others are currently in development.

- **`ADC`** – this module provides supporting functions for using the on-board ADC peripheral.
- **`ISR`** – this module provides supporting functions for declaring ISRs that may also be used by other modules of the API in cases where a conflict might exist when a user wishes to declare his/her own ISR but also happens to be used by a module in the API.
- **`LCD`** – this module provides supporting functions for using the on-board LCD display.
- **`LED`** – this module provides supporting functions for using the on-board LEDs.
- **`PSXC`** – this module provides supporting functions for communicating with a PS2-type (Playstation 2) controller attached via the on-board PS2 controller connector.
- **`SPI`** – this module provides supporting functions for using the *serial peripheral interface* on the MCU.
- **`SPKR`** – this module provides supporting functions for generating audible tones via the on-board speaker.

- **`STEP`** – this module provides supporting functions for controlling the CEENBoT's *stepper motors*.
- **`SWATCH`** – this module provides supporting functions for using the *stopwatch* module, which can be used to measure time in units of **`10us/tick`**.
- **`TINY`** – this module provides supporting functions for using peripherals that are under direct control of a secondary *supporting* MCU, which happens to be the **`ATtiny48`**, hence the name '**`TINY`**' for the module. The **`TINY`** can be used to acquire the state of the on-board push-button switches, control any attached RC servos, and acquire the state of the on-board Infrared (IR) sensors on the CEENBoT.
- **`TMRSRVC`** – this module provides supporting functions for *millisecond* accurate timing services.
- **`UART`** – this module provides supporting functions for using the on-board USARTS in *asynchronous* mode ONLY.
- **`USONIC`** – this module provides supporting functions for using *Ping Ultrasonic senor* from *PARALLAX*.
- **`I2C`** – this module provides supporting functions for operating the ATmega324's I2C interface as both *master* and *slave*. Please note that the *slave* operation has not been thoroughly tested as most users implement I2C as *master* to control some other slave device or peripheral.

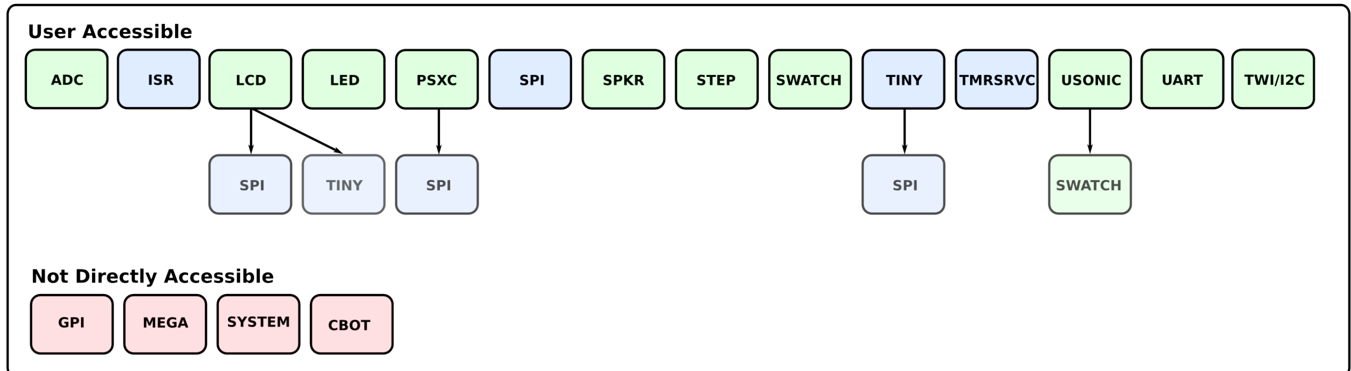
More modules are being created as development continues.

> **Note:** *Only those modules deemed useful to the end-user are given in the list above and covered in this document. There are additional internal supporting modules whose details are not within the scope of this document. These additional 'API internals' are documented elsewhere. At the time this document was being written, this additional information remains to be written.*

It should be noted that except for a few modules which are automatically started internally by the API most of these modules are disabled by default. That is, if you wish to make use of a specific feature or peripheral by way of the API, *you* must start up the module by a specified *open* function. In fact, each subsystem module supports a pair of *open/close* functions used for initializing and acquiring necessary resources for the operation of the module, and releasing and relinquishing these resources when they're no longer needed. Under most circumstances, however, you'll only be concerned with starting up the module, and not with closing it.

Each time a CEENBoT-API-based program starts, the following modules are automatically started: these are the **`SPI`**, **`TINY`**, and **`TMRSRVC`** modules because they are critical to the operation of the API. All other modules must be manually started by the appropriate *open* function call.

The figure below gives a wide overview of the different *services* supplied by the CEENBoT-API. Modules shown in *blue* denote modules that are started by default, while *green* denote those modules which must be manually started by the user. The figure also shows the inherent *dependencies* that bind some modules to others. It can be seen that the **`LCD`**, **`PSXC`**, and **`TINY`** modules depend on the **`SPI`** (which is started by default). Note the *ultrasonic* (**`USONIC`**) module depends on the *stopwatch* (**`SWATCH`**) to measure the pulse width of the 'echo' for the *ping* travel time. However, neither the ultrasonic module nor the stopwatch are started by default, which means if services from these modules are needed, it is the responsibility of the end-user to start both of these modules before invoking any services from them.



The figure also shows the additional internal modules that was alluded to earlier in this document. However, to be thorough their function is summarized below:

- **GPI** – this module implements the GPI-layer (*Graphical Programming Interface*) to support the graphical programming environment that runs on the PC Host. This module is not meant to be used directly by the end-user when writing CEENBoT-API programs.

- **MEGA** – this module provides support functions for manipulating features of the **ATmega324** microcontroller.

- **SYSTEM** (or **SYS**) – this module provides *system-wide* functions used by pretty much all the other modules. It keeps internal state of which modules are open or closed. If a dependency exist, modules use **SYS** to check if dependencies have been opened.

- **CBOT** – this module provides CEENBoT-wide functions. Presently there's not much going on here. Initialization functions for the CEENBoT itself reside here, but these are internally invoked by the API.

Opening and Closing of Subsystem Modules

As already discussed the CEENBoT-API is divided into *subsystem modules*, with each module encapsulating a set of functions that control a specific feature or peripheral. Some modules are started by default – once again, these modules are **TINY**, **SPI**, and **TMRSRVC**. If the user wishes to take advantage of the services provided by any other module, it is the responsibility of the end-user to manually start them.

Each module supports a pair of *open/close* functions. You call the appropriate *open* function to start it, and when you're done, you *close* it. Under most circumstances, however, once you open a subsystem module, there really isn't a need to close it unless the module is tying up a resource you wish to use in a different way. You must look at the function reference for the particular module in question to verify what the *hardware dependencies* might be to determine if there is a need to close a module in question.

Needless to say, the procedure is the *same* for all modules, so for the sake of my sanity in having to explain the same procedure over and over for each module, let us review this process using an example or two.

Procedure for Opening Modules Before Use

All modules have a corresponding set of *open/close* functions prefixed with the name used to identify the module – for example `LCD_open()` opens the **LCD** module, while `STEPPER_open()` module opens the **STEP** subsystem module, etc. All the corresponding *open* functions return a variable of type `SUBSYS_OPENSTAT`, which stands for *subsystem open status*. This *type* is actually a structure with the following form:

```
typedef struct SUBSYS_OPENSTAT_TYPE {

    SUBSYS       subsys;
    SUBSYS_ST    state;

} SUBSYS_OPENSTAT;
```

Starting from the second field entry, `state`, can have two enumerated constants, being `SUBSYS_OPEN`, or `SUBSYS_CLOSED`. You check this field to verify whether an attempt to *open* a module succeeded. The first field you'd want to check ONLY if a problem occurred – for example, state turned out to be `SUBSYS_CLOSED`. when you were expecting `SUBSYS_OPEN`. This field identifies *which* subsystem yielded a problem. For example, "*you requested to use the USONIC module, but I depend on SWATCH, which isn't open just yet, so I'm going to fail and deny your request to open USONIC – thank you.*"

A variable of type SUBSYS as in the structure above can take on one of the following enumerated constants:

SUBSYS_CPU0 – This refers to the *primary* MCU (ATmega324).
SUBSYS_CPU1 – This refers to the *secondary* MCU (ATtiny48).
SUBSYS_SPI – This refers to the SPI subsystem module.
SUBSYS_PSXC – This refers to the PLAYSTATION CONTROLLER subsystem module.
SUBSYS_LCD – This refers to the LCD display subsystem module.
SUBSYS_LED – This refers to the LED subsystem module.
SUBSYS_STEPPER – This refers to the STEPPER subsystem module.
SUBSYS_TMRSRVC – This refers to the TIMER SERVICE subsystem module.
SUBSYS_SPKR – This refers to the TONE GENERATOR subsystem module.
SUBSYS_BEEP – This refers to the BEEP GENERATOR subsystem module.
SUBSYS_SWATCH – This refers to the STOPWATCH service subsystem module.
SUBSYS_USONIC – This refers to the ULTRASONIC subsystem module.
SUBSYS_NA – No subsystem being implied.

The following example shows what should be the typical procedure for opening subsystem modules. The example shows how to open the *ultrasonic* module, which from the diagram previously shown, has a dependency on the *stopwatch* service module.

```
#include "capi324v221.h"

void CBOT_main( void )
{

  SUBSYS_OPENSTAT ops_swatch, ops_usonic;

  // Open the stopwatch module first.
  ops_swatch = STOPWATCH_open();

  // Check that things are good before proceeding.
  if ( ops_swatch.state != SUBSYS_OPEN )
  {

         // ... PANIC BUTTON!...

  } // end if()

  // If no error, continue...
  ops_usonic = USONIC_open();

  // Check that things are good.
  if ( ops_usonic.state != SUBSYS_OPEN )
  {

         // ... PANIC BUTTON!...

  } // end if()
```

(*Continued on next page*)

(*Continued from previous page*)

```
  // If no errors, then do whatever it is you need to do with
  // these modules.
  if ( ops_usonic == SUBSYS_OPEN && ops_swatch == SUBSYS_OPEN )
  {

         // ... invoke functions from USONIC and/or SWATCH.

  } // end if()

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

} // end CBOT_main()
```

Do you have to go through all this procedure when opening modules? Not really. You can just do what I do and assume that they always open when you instruct the API to do so. The same approach is shown below:

```
#include "capi324v221.h"

void CBOT_main( void )
{

  SUBSYS_OPENSTAT ops;

  // Open both modules -- assume they open without problems.
  // here 'ops' is used for inspecting the value returned during
  // a simulation while debugging.
  ops = STOPWATCH_open();
  ops = USONIC_open();


  // Okay... start calling USONIC or SWATCH functions.


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

} // end CBOT_main()
```

In the above example we check nothing and assume that opening of the stopwatch and the ultrasonic module are successful. However, we store the result in a temporary variable so that we can observe it if we end up debugging in a simulated environment.

It is up to you how much *safety* you want to implement into your programs. The main point here is that opening of CEENBoT-API modules all proceed in the same way.

Closing Subsystem Modules & Modular Hardware Dependencies

Why close? …and is it necessary? Well, it is *not* necessary to close a module once open. Under most circumstances if you open up a subsystem module, you will need it for the duration of your program so there's no need to close it. However, there are times when resources taken up by a subsystem module is needed. For example, a *hardware dependency* for the ultrasonic module (**`USONIC`**) uses I/O pin **`PA4`** on the **`ATmega324`** MCU. What if, for whatever reason, you need to use this pin for something else? (*just pretend that you do*). Then you'd have to close the ultrasonic module so that you can take **`PA4`** back.

To close a subsystem module you simply call the corresponding *close* function. These functions take no arguments, nor they return any. You just call the function and you're done. For example:

```
#include "capi324v221.h"

void CBOT_main( void )
{

  SUBSYS_OPENSTAT ops;

  // Open both modules -- assume they open without problems.
  // here 'ops' is used for inspecting the value return during
  // a simulation while debugging.
  ops = STOPWATCH_open();
  ops = USONIC_open();


  // Take a 'sonar' measurement, etc.


  // We're done.
  USONIC_close();     // Close the USONIC module.
  STOPWATCH_close();  // Close the STOPWATCH module.


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

} // end CBOT_main()
```

Note: Make sure you look at the function reference for the respective subsystem module you wish to make use of and check out both *module dependencies* and *hardware dependencies* so that you are aware of what resources you gain and loose as a result of opening the respective subsystem module.

Chapter 2: The CAPI (CEENBoT-API) Subsystem Module

> The **`CAPI`** subsystem module provides general services across the board (API) in general. Presently, this means the ability to obtain the revision info of the API and a random seed value that is generated upon start up.

Module at a Glance

Description

The **CAPI** subsystem module provides services available to all other modules and user's programs across the board. Presently there are only two functions exposed by this module meant to obtain the revision number of the API and an auto-generated random seed value that is generated upon start-up that can be used with `srand()` to set a new random seed value which can be useful in generating unique pseudo-random sequences when `rand()` is called.

Modular Dependencies

The **CAPI** subsystem module has no modular dependencies. Functions are always available by default. Nothing to open; nothing to close.

Hardware Dependencies

The **CAPI** subsystem module has no hardware dependencies.

Function List Summary

- `CAPI_get_revision()` – Used to obtain the revision number of the API.
- `CAPI_get_seed_val()` – Used to obtain random seed value auto-generated upon start-up.

Function Reference

The `CAPI_get_revision()` Function

Format:

```
void CAPI_get_revision( CAPI_REV *pRev )
```

Description:

Function can be used to obtain the revision number of the API. The information is stored in a structure of type `CAPI_REV` which has the following form:

```
// Custom type declaration for obtaining the current revision of the API.
typedef struct CAPI_REV_TYPE {

                                 // Print format: vX.XX.XXXc
    unsigned short int major;   // Major revision number (1 or more digits).
    unsigned short int minor;   // Minor revision number (2 digits exactly).
    unsigned short int build;   // Build number          (3 digits exactly).
    char status;                // Revision status       (1 character exactly).

} CAPI_REV;
```

Notice the structure has four fields, which contain the *major*, *minor*, *build* numbers along with a revision *status* in the form of a character. (See the 'Example' section for usage). Note that the format of the revision numbers *always* take the form of:

vM.mm.BBBc

Where 'M' is the *major* number and it contains 1 or more digits. 'm' is the *minor* number and it ALWAYS consists of two digits only. 'B' is the build number and it ALWAYS consists of 3 digits. Finally, 'c' is the revision status character and is ALWAYS a *single* character and it must be printed as a character (not an integer).

Input Arguments:

`pRev` – You must pass to this argument THE ADDRESS of a structure of type `CAPI_REV`. The structure will then be populated with the values corresponding to the current API revision number.

Example:

```
// Somewhere in your program....
CAPI_REV api_rev;

// Get the current API revision.
CAPI_get_revision( &api_rev );

LCD_open();

// Print the revision in the proper format: vX.XX.XXXc
LCD_printf( "API-Rev: v%d.%02d.%03d%c\n", api_rev.major,
                                          api_rev.minor,
                                          api_rev.build,
                                          api_rev.status );
```

The `CAPI_get_seed_val()` Function

Format:

```
unsigned short int CAPI_get_seed_val( void )
```

Description:

This function can be used to obtain a *new* seed value that can then be fed to the '`srand()`' function to generate a new pseudo-random sequence each time `rand()` is called. The seed value is always stored internally and generated when the CEENBoT is powered.

Input Arguments:

None.

Returns:

Returns an integer value of type `unsigned short int`. This value can be then 'fed' to `srand()`, to set the new seed value.

Example:

```
#include "capi324v221.h"

#include<stdlib.h>

void CBOT_main( void )
{

    unsigned short int seed = 0;
    unsigned short int rand_val = 0;

    // Get new seed value.
    seed = CAPI_get_seed_val();

    // Set the new seed.
    srand( seed );

    // Start obtaining random values...
    rand_val = rand();

    // Etc...

    // Never leave.
    while( 1 );

}
```

Chapter 3: The ADC Subsystem Module

> The **ADC** subsystem module greatly simplifies sampling of analog signals by way of the embedded Analog-to-Digital Converter present on-board the MCU.

Module at a Glance

Description

The ATmega324 contains a 10-bit Analog-to-Digital Converter. Functions exposed by the **ADC** subsystem module greatly facilitate the use of this internal embedded peripheral.

Modular Dependencies

The **ADC** module *must* be manually opened by the user. It has no other modular dependencies.

Hardware Dependencies

The following *potential* hardware dependencies may exist. That means, the I/O pins listed below constitutes the list of pins that *may* be acquired by the ADC subsystem module, *if* the user selects the respective I/O pin as an *input channel* for sampling data. That is, only ONE of the I/O pins listed below will be under control of the ADC at the discretion and *selection* of the user. These pins are:

- **I/O Port pin PA0 on PORTA (ADC0 channel) RESERVED**
- **I/O Port pin PA1 on PORTA (ADC1 channel) RESERVED**
- **I/O Port pin PA2 on PORTA (ADC2 channel) RESERVED**
- I/O Port pin **PA3** on **PORTA** (**ADC3** channel, available via header **J3**, pin **1**)
- I/O Port pin **PA4** on **PORTA** (**ADC4** channel, available via header **J3**, pin **2**)
- I/O Port pin **PA5** on **PARTA** (**ADC5** channel, available via header **J3**, pin **3**)
- I/O Port pin **PA6** on **PORTA** (**ADC6** channel, available via header **J3**, pin **4**)
- I/O Port pin **PA7** on **PORTA** (**ADC7** channel, available via header **J3**, pin **5**)

> **Note:** ADC channels **0**, **1** and **2** are *reserved* and used for monitoring voltage levels of the power subsystem of the CEENBoT (e.g., *Battery current*, *Battery Voltage*, and *Power Supply Voltage*). Consult the schematic for specifics. This means only **ADC** channels **3-7** are *free* for your use. The phrase "*header J3"* refers to the connector labeled **J3** on the CEENBoT's controller board, next to the 3 push-buttons.

Function List Summary

- `ADC_open()` – Open the ADC subsystem module for use.
- `ADC_close()` – Close the ADC subsystem module when no longer needed.
- `ADC_set_channel()` – Sets the *input analog channel* which will be sampled.
- `ADC_set_VREF()` – Sets the *voltage reference* used by the ADC.
- `ADC_sample()` – Take a *single* sample on the currently selected *channel* returning a 10-bit digital representation.

Function Reference

The `ADC_open()` Function

Format:

```
SUBSYS_OPENSTAT ADC_open( void )
```

Description:

Function acquires and initializes resources needed for operation of the **ADC** subsystem module. You *must* call this function first with a successful 'open' before invoking any other function provided by this module.

Returns:

Returns a structure of type `SUBSYS_OPENSTAT` whose field entries indicate the status of the open request and the subsystem that resulted in an error (when, and *if* an error occurs). See *Chapter 1, Procedure for Opening Modules Before Use* for examples on how to handle the *open status*.

Example:

```
#include "capi324v221.h"

void CBOT_main( void )
{

    SUBSYS_OPENSTAT opstat;

    // Open the ADC module.
    opstat = ADC_open();

    if ( opstat.state == SUBSYS_OPEN )
    {

        // ... DO ADC STUFF ...

    } // end if()

} // end CBOT_main()
```

The `ADC_close()` Function

Format:

```
void ADC_close( void )
```

Description:

Function deallocates and releases resources being used by the **ADC** subsystem module. No other functions should be invoked once the subsystem module is closed.

The `ADC_set_channel()` Function

Format:

```
void ADC_set_channel( ADC_CHAN which )
```

Description:

This function allows you to set the *input channel* that will be used for subsequent samples upon invocation of `ADC_sample()` function (discussed in this document). The channel will remain set until the user changes it by invoking `ADC_set_channel()` to another channel.

Input Arguments:

`which` – Specifies the source of the *analog signal* for conversion. Must be one of the following enumerated constants:

`ADC_CHAN0`	– Specify **ADC0** channel.
`ADC_CHAN1`	– Specify **ADC1** channel.
`ADC_CHAN2`	– Specify **ADC2** channel.
`ADC_CHAN3`	– Specify **ADC3** channel.
`ADC_CHAN4`	– Specify **ADC4** channel.
`ADC_CHAN5`	– Specify **ADC5** channel.
`ADC_CHAN6`	– Specify **ADC6** channel.
`ADC_CHAN7`	– Specify **ADC7** channel.
`ADC_CHAN_VBG`	– Specify the *internal Bandgap Voltage reference* channel (1.1V).
`ADC_CHAN_GND`	– Specify the *internal ground Voltage reference* (0V).

> **Note:** Selecting `ADC_CHAN_GND` essentially allows you to disconnect the **ADC** from any external channels. This is the default channel when the **ADC** subsystem module is opened (or re-opened) for the first time.

Example:

See the example for `ADC_sample()` function.

The `ADC_set_VREF()` Function

Format:

```
void ADC_set_VREF( ADC_VREF which )
```

Description:

This function allows you to set the *voltage reference* that will be used by the ADC. The voltage reference does two things:

- It determines what voltage range can be digitally represented and sampled.
- It determines what the smallest resolution (in LSBs) can be achieved with the ADC.

The *digital code* representation of an analog signal is given by the following relation:

$$ADC = \frac{V_{\text{in}} \cdot 1024}{V_{\text{REF}}} \text{, where } 0 \le ADC \le 2^{10} - 1 = 1023$$

Consequently, the *minimum* digital voltage representation (or LSB) is given by:

$$V_{\text{LSB}} = \frac{V_{\text{REF}}}{1024} \quad \text{where the maximum possible representation is given by} \quad V_{\max} = V_{\text{REF}} \cdot \left(\frac{1023}{1024} \right)$$

Keep in mind that **`1023`** is the maximum *binary code* representation for the 10-bit ADC.

Input Arguments:

which – Must be one of the following enumerated constants:

`ADC_VREF_AREF`	– Specifies internal **`VREF`** to be turned OFF (default upon opening module).
`ADC_VREF_AVCC`	– Specifies **`AVCC`** pin voltage as **`VREF`** (use **`VREF=5V`**).
`ADC_VREF_1P1V`	– Specifies to use the *internal bandgap voltage reference* as **`VREF`** (use **`VREF=1.1V`**).
`ADC_VREF_2P56V`	– Specifies to use the *internal voltage reference* as **`VREF`** (use **`VREF=2.56V`**).

Example:

See the example for `ADC_sample()` function.

The `ADC_sample()` Function

Format:

```
ADC_SAMPLE ADC_sample( void )
```

Description:

This function will take a 'sample' of the analog signal of the *currently selected ADC channel* and return the 10-bit digital representation of the same.

Returns:

Returns a value of type `ADC_SAMPLE`, which presently defaults to an `unsigned short int`. This value corresponds to the 10-bit digital representation of the analog signal, taken from the currently selected ADC channel. To convert this 10-bit *code* to its voltage representation use:

$$V_{\text{in}} = ADC \cdot \left(\frac{V_{\text{REF}}}{1024} \right)$$

Example:

```
// Assume the ADC is properly opened.
// Pretend we're measuring a voltage on ADC channel 3
// that can range between 0V to 5V.

ADC_SAMPLE sample;  // Store CODE.

float voltage;      // Store Voltage representation of 'CODE'.

// Set the Voltage Reference first so VREF=5V.
ADC_set_VREF( ADC_VREF_AVCC );

// Set the channel we will sample from.
ADC_set_channel( ADC_CHAN3 );

// Now sample it!
sample = ADC_sample();

// Convert to meaningful voltage value.
voltage = sample * ( 5.0 / 1024 );
```

Chapter 4: The ISR Subsystem Module

The `ISR` subsystem module gives user's of the API to dynamically assign *interrupt service routines* or ISRs on the fly, thereby allowing the sharing of ISR resources between the user and the API as needed.

Module at a Glance

Description & Rationale

The CEENBoT-API is built by using the AVR-GCC GNU compiler tool chain. When the a user wishes to write an *interrupt service routine* or ISR and associate this routine with a specific interrupt, the user must declare the ISR by using the `ISR()` macro as outlined in the **`AVR-Libc`** documentation for the AVR-GCC GNU compiler tool chain. For example, when using one of the timers, say, **`Timer1`**, the user might use the *timer compare match* interrupt when operating the timer in CTC mode:

```
ISR( TIMER1_COMPA_vect )
{

    // ... code here ...

}
```

Now, the problem here is that some *subsystem modules* available through the CEEN-BoT API may use resources that also necessitate the need to declare ISRs internally. The problem is that if the user wishes to use an MCU resource that is also used by a module of the API, and he/she wishes to write his/her own ISR routine, there will be a *conflict* and an error will be issued by the compiler because the ISR is already declared and defined internally by the CEENBoT-API.

Not all ISRs that are available to the user are permanently 'hi-jacked' by the CEENBoT-API – only the ISRs falling in the following categories are hi-jacked by the CEENBoT-API because they are used for the operation of some of the subsystem modules – but only, if such modules are opened by the user. These are:

- Any *timer-related* interrupts: **`Timer0`**, **`Timer1`** and **`Timer2`**
- Any *pin-change* interrupts: Groups **`0`**, **`1`**, **`2`**, and **`3`**.

The only exception is **`Timer0`**. You CANNOT override any interrupts assigned to **`Timer0`** as **`Timer0`** is reserved for use by the API. However interrupts that refer to any of the other resources listed above are 'optionally' available. “optionally available” means that as long as a module that needs that resource isn't OPEN, the ISR that such resource might rely on, is available for *your* use. However – once you OPEN the related device which may also need the *same* ISR, then the subsystem module will hi-jack it from you and take over. So your responsibility lies in checking out the *hardware dependencies* section for any subsystem module you may be opening, and be aware of what ISRs (if any) will/may be taken away from you if you do choose to open said subsystem module. That means, you have to be willing to share an ISR with the CEENBoT-API.

How do you do that? By using the `CBOT_ISR()` macro to declare and define the interrupt service instead of the standard `ISR()` macro furnished by **`AVR-Libc`**. You then *register* your ISR with the **`ISR`** subsystem module. For example, the above ISR using the TIMER1_COMPA_vect ISR vector is a *timer-related* interrupt – so you *have* to share – therefore, assuming no other subsystem module is using *your* ISR, then you would do the following:

```
CBOT_ISR( myTimerISR ); // Prototype.

CBOT_ISR( myTimerISR ) // Definition.
{

    // ... code here ...

}
```

(*Continued on next page*)

(*Continued from previous page*)

Then somewhere in `CBOT_main()`, you would do the following:

```
void CBOT_main( void )
{

   // Register our ISR and attach it to 'TIMER1_COMPA_vect' interrupt.
   ISR_attach( ISR_TIMER1_COMPA_VECT, myTimerISR );

   // ...blah, blah.

   while( 1 ); // Never leave.

}
```

It's as easy as that.

However, now suppose that after you do *blah, blah*, you decide to use the *stopwatch subsystem module* by invoking `STOPWATCH_open()`. This subsystem module also uses **`Timer1`** in CTC mode, so the ISR you just registered will be hi-jacked by this module and your ISR will no longer take effect. So you need to look at documentation and be fully aware when such dependencies exist.

> **Note:** Use `CBOT_ISR()` to declare your own ISRs related to any of the *timers* or *pin-change* interrupts.

> **Note:** Use the standard `ISR()` to declare all other ISRs.

Why have two ISR declaration macros? Why the need for `CBOT_ISR()` *and* `ISR()`? The reason behind this is because `CBOT_ISR()` works by *pre-defining* an ISR hook for any supported ISRs by the MCU. If a 'hook' is supplied for *every* possible supported ISR by the MCU it would end up consuming too much precious memory. It is rare for a user to have a need to use *all* ISRs. Therefore, a choice was made to only implement `CBOT_ISR()` for those ISRs that might present a conflict and control the sharing of these ISRs between the user and the API. This conserves memory.

Modular Dependencies

The **`ISR`** subsystem module is already *open by default* since it constitutes a critical component of the API.

> **Note:** The **`ISR`** subsystem module is *open by default*.

Hardware Dependencies

The ISR subsystem module by itself has no hardware dependencies. However, you should verify the hardware dependencies for any particular modules to determine if an ISR will eventually be 'hi-jacked' by such module once it is opened.

Function List Summary

- `ISR_open()` – Opens the ISR module for use.
- `ISR_close()` – Closes the ISR module after use.
- `CBOT_ISR()` – Used to declare an ISR which may be *shared* with other subsystem modules.
- `ISR_attach()` – Used to dynamically register a custom ISR previously declared with `CBOT_ISR()`.

Function Reference

The `ISR_open()` Function

Format:

```
SUBSYS_OPENSTAT ISR_open( void )
```

Description:

Function acquires and initializes resources needed for operation of the ISR subsystem module. You *must* call this function first with a successful 'open' before invoking any other function provided by this module.

Returns:

Returns a structure of type `SUBSYS_OPENSTAT` whose field entries indicate the status of the open request and the subsystem that resulted in an error (when, and *if* an error occurs). See *Chapter 1, Procedure for Opening Modules Before Use* for examples on how to handle the *open status*.

> **Note:** Please note the **`ISR`** module is open by default.

The `ISR_close()` Function

Format:

```
void ISR_close( void )
```

Description:

Function deallocates and releases resources being used by the ISR subsystem module. No other functions should be invoked once the subsystem module is closed.

> **Note:** You shouldn't attempt to *close* the **`ISR`** module – it is a *critical* component of the API.

The `CBOT_ISR()` *Macro*-Function

Format:

`CBOT_ISR( ISR_name )`

Description:

This macro-function *replaces* the use of the standard `ISR()` macro to declare and define custom *interrupt service routines* for ONLY those ISRs that are shared between the API and the user. All other interrupt services can be declared with the standard `ISR()`. Essentially any ISRs that relate to the following:

- Any *timer-related* interrupts: **`Timer0`**, **`Timer1`**, **`Timer2`**
- Any *pin-change related* interrupts: Group **`0`**, **`1`**, **`2`**, and **`3`**

Are *shared* – and thus such interrupts *must* be declared via `CBOT_ISR()` and *not* via `ISR()`.

> **Note:** **`Timer0`** related ISRs cannot be used as **`Timer0`** is reserved for use by the API.

Input Arguments:

`ISR_name` – This is the name of *your* ISR. It can be anything that can constitute a *legal function name*.

Example:

The following 'snippet' example declares an ISR to be associated with *Timer2*.

```
// Prototype
CBOT_ISR( Timer2_CTC_ISR );

// Definition
CBOT_ISR( Timer2_CTC_ISR ) {

    // ... code here ...

}

void CBOT_main( void ) {

    // 1) Initialize Timer2 as needed (for CTC mode).

    // 2) Register our ISR.
    ISR_attach( ISR_TIMER2_COMPA_VECT, Timer2_CTC_ISR );

    // 3) Enable Timer2 interrupts and let the show begin.

}
```

Note that you have to both *declare* [a prototype] and *define* the ISR, just like you would any other function!

The `ISR_attach()` Function

Format:

```
CBOT_ISR_FUNC_PTR ISR_attach( ISR_VECT vect, CBOT_ISR_FUNC_PTR isr_function )
```

Description:

Use this function to *register* your custom ISR (declared via `CBOT_ISR()` macro) with the **`ISR`** subsystem module. Note that this only registers your ISR. Your ISR will be *triggered* once the interrupts for the corresponding device are enabled somehow.

Input Arguments:

`vect` – This specifies which *vector* your ISR will be associated with. It must be one of the following enumerated constants:

`ISR_PCINT0_VECT`	– Pin-change interrupt Group 0
`ISR_PCINT1_VECT`	– Pin-change interrupt Group 1
`ISR_PCINT2_VECT`	– Pin-change interrupt Group 2
`ISR_PCINT3_VECT`	– Pin-change interrupt Group 3
`ISR_TIMER0_COMPA_VECT`	– **`Timer0`** Compare-Match A interrupt
`ISR_TIMER0_COMPB_VECT`	– **`Timer0`** Compare-Match B interrupt
`ISR_TIMER0_OVF_VECT`	– **`Timer0`** Overflow interrupt
`ISR_TIMER1_COMPA_VECT`	– **`Timer1`** Compare-Match A interrupt
`ISR_TIMER1_COMPB_VECT`	– **`Timer1`** Compare-Match B interrupt
`ISR_TIMER1_OVF_VECT`	– **`Timer1`** Overflow interrupt
`ISR_TIMER1_CAPT_VECT`	– **`Timer1`** Capture interrupt
`ISR_TIMER2_COMPA_VECT`	– **`Timer2`** Compare-Match A interrupt
`ISR_TIMER2_COMPB_VECT`	– **`Timer2`** Compare-Match B interrupt
`ISR_TIMER2_OVF_VECT`	– **`Timer2`** Overflow interrupt

> **Note:** Don't see the interrupt vector you're looking for? Then that means you must declare your ISR the normal way – by using the `ISR()` macro. See the AVR-Libc documentation (interrupts section) on how to declare such interrupts. The interrupt vectors that you see above are the ones where a conflict can potentially exist because the API might use them. The *very* purpose behind the existence of the ISR module is to circumvent this potential conflict.

Example:

The following example shows a code-snippet for declaring ISRs. Here two ISRs are declared – the first one for the compare-match interrupt for **Timer2**. This interrupt, being *timer-related* must be 'shared' with the API and as a result, we declare this interrupt via CBOT_ISR() macro. Note that any ISRs declared via CBOT_ISR() macro also require a *prototype*. The second ISR declared is for the external interrupt (**INT0**). This interrupt is neither pin-change related nor timer-related. Therefore, we can declare this interrupt the *normal* way – that is, using the ISR() macro as explained in the AVR-Libc documentation. Note that any ISRs declared via the ISR() macro DO NOT require a *prototype*.

```
// Prototype
CBOT_ISR( myTimer2_ISR );

// Definition
CBOT_ISR( myTimer2_ISR ) {

    // ... code here ...

}

ISR( INT0_vect ) {

    // ... code here ...

}

void CBOT_main( void ) {

    // 1) Initialize Timer2 as needed (for CTC mode).

    // 2) Register our ISR.
    ISR_attach( ISR_TIMER2_COMPA_VECT, Timer2_CTC_ISR );

    // 3) Enable Timer2 interrupts and let the show begin.
    // 4) Enable External Interrupts and let the show begin.

}
```

Chapter 5: The LCD Subsystem Module

This chapter covers the LCD subsystem module and function reference.

Module at a Glance

Description

The **LCD** subsystem module provides functional services for operating the on-board LCD display. Presently this consists of a 128x64 pixel backlit liquid crystal display. In addition, the function list exposed by the **LCD** module provides *character-based* control. *Pixel-based* control is not yet implemented, but *may* be implemented in future releases of the API.

The current LCD display device can accommodate 4 lines of text with a maximum capacity of approximately 21 characters per line (or *row*). The coordinate system used uses the top-most line as line number **3**, while the bottom-most line is line **0**. Essentially, the origin of the display **(0,0)** actually refers to the bottom left-hand corner of the screen, and not the top left-hand corner as it should be. Thus, when writing lines to the LCD in order from top to bottom, you start at line **3**, then **2**, then **1**, and finish at **0**, in *that* order.

> **Note:** The reasons for this awkwardness in the coordinate system used for the LCD are mostly historical. Some of the source code implemented in the LCD module was ported from a different source. The coordinate system may be changed in future API releases.

Modular Dependencies

The **LCD** module *must* be manually opened. It has the following *modular* dependencies:

- **SPI** - The LCD is an SPI device (open by default)
- **TINY** - Back light controlled by the **ATtiny** (open by default)

Hardware Dependencies

- I/O port pin **PB3** on **PORTB**

Function List Summary

- **LCD_open()** - Open and initialize the **LCD** subsystem module.
- **LCD_close()** - Close and release the **LCD** subsystem module.
- **LCD_clear()** - Clear the contents of the LCD display.
- **LCD_putchar()** - Write a single character to the next logical location on the LCD.
- **LCD_putchar_RC()** - Write a single character at the specified *row* and *column* (RC).
- **LCD_set_RC()** - Set the logical location (*row* and *column*) for all subsequent LCD writes.
- **LCD_set_backlight()** - Set the *backlight* intensity.
- **LCD_printf()** - The “all-familiar” printf() function – yes, the *same* one you're used to.... almost.
- **LCD_printf_RC()** - The “all-familiar” printf() function, where the *row* and *column* locations can be specified.
- **LCD_printf_PGM()** - The “all-familiar” printf() function, with the distinction that constant strings are stored in *Program Memory* instead of SRAM.
- **LCD_printf_RC_PGM()** - The “all-familiar” printf() function, where the *row* and *column* locations can be specified and with the difference that constant strings are stored in *Program Memory* instead of SRAM.

- `LCD_draw_xbm()` – Draw X-bitmap image to the LCD display. The xbm data structure must be saved and structure in a *very particular* manner.

- `LCD_register_lcd_change_notify()` – Allows the user to register a *callback* function that is triggered each time the contents of the LCD are modified (re-drawn in some way). Allowing the user to re-paint any text/graphics as needed, and (if) needed.

Function Reference

The `LCD_open()` Function

Format:

```
SUBSYS_OPENSTAT LCD_open( void )
```

Description:

Function acquires and initializes resources needed for operation of the **LCD**. You *must* call this function first with a successful 'open' before invoking any other function provided by this module.

Returns:

Returns a structure of type `SUBSYS_OPENSTAT` whose field entries indicate the status of the open request and the subsystem that resulted in an error (when, and if an error occurs). See *Chapter 1, Procedure for Opening Modules Before Use* for examples on how to handle the *open status*.

Example:

```
#include "capi324v221.h"

void CBOT_main( void )
{

  SUBSYS_OPENSTAT opstat;

  // Open the LCD module.
  opstat = LCD_open();

  if ( opstat.state == SUBSYS_OPEN )
  {

         // ... DO LCD STUFF ...

  }// end if()

} // end CBOT_main()
```

The `LCD_close()` Function

Format:

```
void LCD_close( void )
```

Description:

Function deallocates and releases resources being used by the **LCD** subsystem module. No other functions should be invoked once the subsystem module is closed.

The `LCD_clear()` Function

Format:

```
void LCD_clear( void )
```

Description:

Use this function to clear the contents on the LCD display. The LCD will be 'blanked'.

The `LCD_putchar()` Function

Format:

```
void LCD_putchar( char c )
```

Description:

Function allows a *single character* to be written on the LCD. It will be written on the next logical cursor location awaiting to be written.

Input Arguments:

`c` – The ASCII character to write to the LCD display.

The `LCD_putchar_RC()` Function

Format:

```
void LCD_putchar_RC( unsigned char row, unsigned char col, char c )
```

Description:

Function allows a *single character* to be written on the LCD. It will be written on the *row* and *column* locations specified.

Input Arguments:

`row` – The *row* where the character will be written (**`0 - 3`**, with **`3=top-most line`**, **`0=bottom-most line`**).

`col` – The *column* where the character will be written (**`0 - 21`**).

`c` – The ASCII character to write to the LCD display.

Example:

(*See next page*)

```
unsigned int i;
const char str[] = "Hello, Dolly!";

for ( i = 0; i < 13; i++ )

  // Print on top-most line character-by-character.
  LCD_putchar_RC( 3, i, str[ i ] );
```

The `LCD_set_RC()` Function

Format:

`void LCD_set_RC( unsigned char row, unsigned char col )`

Description:

This function allows you to set the next logical location – *row* and *column* – for all subsequent characters written to the LCD.

Input Arguments:

`row` – The *row* where the character will be written (**`0 – 3`**, with **`3=top-most line`**, **`0=bottom-most line`**).

`col` – The *column* where the character will be written (**`0 – 21`**).

Example:

```
// Assume LCD module is properly opened.

// Set next write to top left-most.
LCD_set_RC( 3, 0 );
LCD_printf( "Hello!" );

// Set next write to bottom left-most.
LCD_set_RC( 0, 0 );
LCD_printf( "Dolly!" );
```

The `LCD_set_backlight()` Function

Format:

`void LCD_set_backlight( unsigned char BL_level )`

Description:

This function allows you to set the *backlight level*, by instructing the **`ATtiny`** (supporting MCU) to set the appropriate pulse-width modulated signal since backlight leveling is under control of the supporting MCU.

(*Continued on next page*)

(*Continued from previous page*)

Input Arguments:

`BL_level` – The backlight level. (**`0 = Backlight OFF, 31 = Backlight FULL brightness`**). The default backlight level is **`10`**.

The `LCD_printf()` Function

Format:

```
void LCD_printf( const char *str_format, ... )
```

Description:

This is your standard `printf()` facility. Being that the API operates in an *embedded environment*, the `printf()` facility is not quite full featured (as you would expect on, say, your PC system).

> **Note:** It is not within the scope of this document to teach you how to use `printf()`. It is assumed that if you know how to program in the C programming language, that you are familiar with the `printf()` facility – which is the *very first* thing ANYONE learning C learns how to use. Consult your favorite C programming book for examples on how `printf()` can be used.

> **Note:** The `printf()` facility is not fully featured. For example, printing of *floating point* values is not enabled by default. This is to save 'code space' since usage of the `printf()` function can result in significant increase in code size. Please see the *Getting Started* guide where the matter of using *floating point* with `printf()` is addressed. There are some *linker options* that must be enabled to use `printf()` with *floating point* values.

Input Arguments:

`str_format` - A constant string with appropriate format specifiers such as `%s`, `%d`, `%x`, and/or escape sequences. etc. Note that not all format specifiers, nor escape sequences are supported.

`...` - Zero or more additional parameters as indicated in the *format string* to be included as part it (the *string*). This is the set of *variadic* parameters that are passed to the function in the traiditional '`printf()`' manner.

Example:

Here's a quick example (*see next page*):

(*from previous page*)

```
// Assume LCD module is properly opened.

unsigned int data = 127;
unsigned int    i = 0;
char                c = 'y';

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

  // Clear the screen on each iteration.
  LCD_clear();

  // Print 'data' as HEX.
  LCD_printf( "data (hex) = 0x%X\n", data++ );

  // Print 'i' as DECIMAL.
  LCD_printf( "i (dec) = %d\n", i );

  // Print 'c' as CHARACTER.
  LCD_printf( "c (char) = %c\n", c);

} // end for()
```

The `LCD_printf_RC()` Function

Format:

`void LCD_printf_RC( unsigned char row, unsigned char col,  const char *str_format, ... )`

Description:

This function works *just* like `LCD_printf()` as documented above with the only exception that you can specify the *row* and *column* (hence 'RC') that the string will be printed on the LCD display.

Input Arguments:

`row` – The *row* where the character will be written
(**`0 - 3`**, with **`3=top-most line`**, **`0=bottom-most line`**).

`col` – The *column* where the character will be written (**`0 - 21`**).

`str_format` - A constant string with appropriate format specifiers such as `%s`, `%d`, `%x`, and/or escape sequences. etc. Note that not all format specifiers, nor escape sequences are supported.

`...` - Zero or more additional parameters as indicated in the *format string* to be included as part it (the *string*). This is the set of *variadic* parameters that are passed to the function in the traiditional '`printf()`' manner.

(*Continued from previous page*)

Example:

```
// Assume the LCD module is appropriately OPEN.

unsigned int i;

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

  // Always print in the same position.
  // NOTE: The '\t' clears everything after the last
  //       character is written.
  LCD_printf_RC( 3, 0, " i = %d\t", i );

} // end for()

// Print at the bottom of the LCD display.
LCD_printf_RC( 0, 0, "Done!" );
```

The `LCD_printf_PGM()` Function

Format:

```
void LCD_printf_PGM( const char *str_format, ... )
```

Description:

This function works *just* like `LCD_printf()` with the exception that the constant string is stored in *Program Memory* (hence 'PGM') instead of SRAM. This is useful because constant strings can quickly eat up SRAM. When using this function you *must* declare your constant string to reside in Program Memory via the `PSTR()` macro (see the 'Example' section below). Other than that – *anything* that you can do with `LCD_printf()`, you can do with *this* function.

Input Arguments:

`str_format` - A constant string with appropriate format specifiers such as `%s`, `%d`, `%x`, and/or escape sequences. etc. Note that not all format specifiers, nor escape sequences are supported. The string *must* be declared to reside in *Program Memory* via the `PSTR()` macro.

`...` - Zero or more additional parameters as indicated in the *format string* to be included as part it (the *string*). This is the set of *variadic* parameters that are passed to the function in the traditional '`printf()`' manner.

(*Continued on next page*)

(*Continued from previous page*)

Example:

```
// Assume the LCD subsystem has been properly opened.

// Somewhere in 'CBOT_main()'.

unsigned short int value = 20;

// Print some string.
LCD_printf_PGM( PSTR( "Hello World!\n" ) );

// Print the value of a variable.
LCD_printf_PGM( PSTR( "Value = %d\n" ), value );
```

Note in particular the usage of the `PSTR()` macro. This is required with the PGM functions.

The `LCD_printf_RC_PGM()` Function

Format:

```
void LCD_printf_RC_PGM( unsigned char row, unsigned char col,  const char *str_format, ... )
```

Description:

This function works *just* like `LCD_printf_RC()` with the exception that the constant string is stored in *Program Memory* (hence 'PGM') instead of SRAM. This is useful because constant strings can quickly eat up SRAM. When using this function you *must* declare your constant string to reside in Program Memory via the `PSTR()` macro (see the 'Example' section below). Other than that – *anything* that you can do with `LCD_printf_RC()`, you can do with *this* function.

Input Arguments:

`row` – The *row* where the character will be written (**`0 – 3`**, with **`3=top-most line`**, **`0=bottom-most line`**).

`col` – The *column* where the character will be written (**`0 – 21`**).

`str_format` - A constant string with appropriate format specifiers such as `%s`, `%d`, `%x`, and/or escape sequences. etc. Note that not all format specifiers, nor escape sequences are supported. The string *must* be declared to reside in *Program Memory* via the `PSTR()` macro.

`...` - Zero or more additional parameters as indicated in the *format string* to be included as part it (the *string*). This is the set of *variadic* parameters that are passed to the function in the traditional '`printf()`' manner.

Example:

```
// Assume the LCD subsystem has been properly opened.

// Somewhere in 'CBOT_main()'.

unsigned short int value = 20;

// Print some string on the top-most line.
LCD_printf_RC_PGM( 3, 0, PSTR( “Hello World!\n” ) );

// Print the value of a variable at the bottom-most line.
LCD_printf_RC_PGM( 0, 0, PSTR( “Value = %d\n” ), value );
```

Once again, notice the usage of the `PSTR()` macro. This is required for all PGM functions.

The `LCD_draw_xbm()` Function

Format:

```
void LCD_draw_xbm( const char *p_xbm_data )
```

Description:

This function is more of an 'experimental' feature. It allows you to draw xbm (X-bitmap) image on the LCD. The XBM file format is an image format that is written in the form a *actual* C source code. The best way to produce an XBM image is to use *GIMP*. The rules are as follows:

- The XBM image *must* be 128x32 pixels – NO EXCEPTIONS!
- Rotate the finish image by 90-degrees clockwise.
- Save.

After saving – open the XBM file and copy and paste the data string onto your source code while making sure the declaration is modified in the following form:

```
const char <array_name> [] PROGMEM = { 0x33, 0x44, 0x24, ..., 0x25, 0x01 };
```

In other words, make sure the array (whatever it might be named) is declared as **`const`**, and with the **`PROGMEM`** attribute as shown so that the data is stored in *Program Memory* (you **DO NOT** have enough SRAM to store an image this size in it!). Furthermore, the array declaration should be declared *globally*. See the 'Example' section.

Input Arguments:

`p_xbm_data` – You must pass to this argument THE ADDRESS of an array of chars containing the bitmap data. This is normally auto-generated by the XBM image program when you save the image into XBM file format. See the 'Example' section.

Example:

- Open the GIMP.
- Set up the image size to be 128x32 pixels.
- Draw whatever it is you're going to draw.
- When finished, rotate your image 90-degrees clockwise.
- Save

Then *open* the XBM file you just saved and copy and paste this info into your source code. Here's an example of the contents of a similar XBM file containing a random image I created on the GIMP. I called the image "DeleteMe.xbm":

```
#define DeleteMe_width 32
#define DeleteMe_height 128
static const char DeleteMe_bits[] PROGMEM = {

   0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
   0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x0e,
   0x00, 0x00, 0xc0, 0x3f, 0x00, 0x00, 0xe0, 0x7f, 0x00, 0x00, 0xf0, 0x7f,
   0x00, 0x00, 0xfc, 0xff, 0x00, 0x00, 0xfc, 0xff, 0x00, 0x00, 0xfe, 0xff,
   0x00, 0x00, 0xff, 0x7f, 0x00, 0x00, 0xff, 0x7f, 0x00, 0x00, 0xff, 0x3f,
   0x00, 0x80, 0xff, 0x1f, 0x00, 0x80, 0xff, 0x0f, 0x00, 0xc0, 0xff, 0x07,
   0x00, 0xc0, 0xff, 0x03, 0x00, 0xc0, 0xff, 0x01, 0x00, 0xe0, 0xff, 0x01,
   0x00, 0xe0, 0xff, 0x01, 0x00, 0xe0, 0xff, 0x00, 0x00, 0xf0, 0xff, 0x00,
   0x00, 0xf0, 0xff, 0x00, 0x00, 0xf0, 0x7f, 0x00, 0x00, 0xf0, 0x7f, 0x00,
   0x00, 0xf0, 0x7f, 0x00, 0x00, 0xf8, 0x7f, 0x00, 0x00, 0xf8, 0x3f, 0x00,
   0x00, 0xf8, 0x3f, 0x00, 0x00, 0xf8, 0x3f, 0x00, 0x00, 0xf8, 0x3f, 0x00,
   0x00, 0xf8, 0x3f, 0x00, 0x00, 0xf8, 0x3f, 0x00, 0x00, 0xf8, 0x7f, 0x00,
   0x00, 0xfc, 0x7f, 0x00, 0x00, 0xfe, 0xff, 0x00, 0x00, 0xfe, 0xff, 0x00,
   0x00, 0xfe, 0xff, 0x00, 0x00, 0xff, 0xff, 0x00, 0x00, 0xff, 0xff, 0x00,
   0x00, 0xff, 0x7f, 0x00, 0x00, 0xff, 0x3f, 0x00, 0x80, 0xff, 0x1f, 0x00,
   0x80, 0xff, 0x07, 0x00, 0x80, 0xff, 0x07, 0x00, 0x80, 0xff, 0x07, 0x00,
   0x80, 0xff, 0x07, 0x00, 0x00, 0xff, 0x07, 0x00, 0x80, 0xff, 0x07, 0x00,
   0x00, 0xff, 0x07, 0x00, 0x00, 0xff, 0x07, 0x00, 0x00, 0xff, 0x07, 0x00,
   0x00, 0xff, 0x0f, 0x00, 0x00, 0xff, 0x0f, 0x00, 0x00, 0xfe, 0x3f, 0x00,
   0x00, 0xff, 0x3f, 0x00, 0x00, 0xff, 0xff, 0x00, 0x80, 0xff, 0xff, 0x00,
   0x80, 0xff, 0xff, 0x00, 0xc0, 0xff, 0xff, 0x01, 0xc0, 0xff, 0xff, 0x01,
   0xc0, 0xff, 0xff, 0x01, 0xc0, 0xff, 0xff, 0x01, 0xc0, 0xff, 0xff, 0x00,
   0xc0, 0xff, 0x7f, 0x00, 0xc0, 0xff, 0x5f, 0x00, 0xc0, 0xff, 0x01, 0x00,
   0xc0, 0xff, 0x03, 0x00, 0xc0, 0xff, 0x07, 0x00, 0xc0, 0xff, 0x0f, 0x00,
   0xe0, 0xff, 0x3f, 0x00, 0xf0, 0xff, 0x7f, 0x00, 0xf0, 0xff, 0x7f, 0x00,
   0xf8, 0xff, 0xff, 0x00, 0xf8, 0xff, 0xff, 0x00, 0xf8, 0xff, 0xff, 0x01,
   0xfc, 0xff, 0xff, 0x00, 0xfc, 0xff, 0xff, 0x00, 0xfc, 0xff, 0xff, 0x00,
   0xfc, 0xff, 0x7f, 0x00, 0xfc, 0xbf, 0x1f, 0x00, 0xf8, 0x3f, 0x00, 0x00,
   0xfc, 0x3f, 0x00, 0x00, 0xf8, 0x3f, 0x00, 0x00, 0xf8, 0x7f, 0x01, 0x00,
   0xf8, 0xff, 0x07, 0x00, 0xf8, 0xff, 0x1f, 0x00, 0xf0, 0xff, 0x7f, 0x00,
   0xf0, 0xff, 0x7f, 0x00, 0xf0, 0xff, 0xff, 0x01, 0xe0, 0xff, 0xff, 0x03,
   0xe0, 0xff, 0xff, 0x07, 0xc0, 0xff, 0xff, 0x07, 0xc0, 0xff, 0xff, 0x0f,
   0x80, 0xff, 0xff, 0x0f, 0x80, 0xff, 0xff, 0x1f, 0x80, 0xff, 0xff, 0x1f,
   0x80, 0xff, 0xff, 0x1f, 0x80, 0xff, 0xff, 0x1f, 0x80, 0xff, 0xff, 0x0f,
   0x00, 0xff, 0xff, 0x07, 0x00, 0xff, 0xff, 0x05, 0x00, 0xff, 0x5f, 0x00,
   0x00, 0xfe, 0x1f, 0x00, 0x00, 0xfe, 0x1f, 0x00, 0x00, 0xfe, 0x3f, 0x00,
   0x00, 0xfc, 0x3f, 0x00, 0x00, 0xfc, 0x7f, 0x00, 0x00, 0xfe, 0x7f, 0x00,
   0x00, 0xfe, 0x7f, 0x00, 0x00, 0xfe, 0x7f, 0x00, 0x00, 0xff, 0x3f, 0x00,
   0x00, 0xff, 0x3f, 0x00, 0x00, 0xff, 0x1f, 0x00, 0x00, 0xff, 0x0f, 0x00,
   0x00, 0xfe, 0x07, 0x00, 0x00, 0xfe, 0x07, 0x00, 0x00, 0xfe, 0x07, 0x00,
   0x00, 0xfc, 0x01, 0x00, 0x00, 0xf0, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00,
   0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
   0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00

};
```

The '`const`' part was added, and the '`char`' was previously '`unsigned char`', and the `PROGMEM` was added. The contents of this structure represent the XBM image data. Provided the LCD subsystem has been properly opened, all you would need to do to display the above image is to do this:

```
LCD_draw_xbm( DeleteMe_bits );        // Draw the XBM image.
```

Supported Escape Sequences

The following escape sequences are supported by any of the `LCD_printf...()` functions:

\n – The *newline* character. Just as with the standard 'printf()' this consists of a new line sequence: *a carriage return* and a *line feed*.

\r – This is a *carriage return* ONLY. Brings the cursor back to the beginning of the line but it doesn't give a *new* line.

\f – This is the *form feed*. It gives a *new* line, but it doesn't bring the cursor back to the beginning of the line.

\e – Causes the contents of the LCD to be cleared. Doing a '`LCD_printf( "\e" )`' yields the same result as calling `LCD_clear()`.

\t – Causes everything that follows the \t to be erased until the end of the *current* line. For example, '`LCD_printf( "Percent: %d%\t", percent_val )`'.

\b – Backspace by one character.

\a – Causes a short 'beep'. For example: '`LCD_printf( "Wrong Answer!\n\a" )`'.

Chapter 6: The LED Subsystem Module

This chapter covers the **LED** subsystem module and function reference.

Module at a Glance

Description

The **LED** subsystem modules provides functional services for controlling the on-board LEDs. Presently the CEENBoT '324 v2.21 platform only support two controllable LEDs, *red* and *green*. The functions in this module allow you to manipulate these LEDs in various convenient ways.

Modular Dependencies

The **LED** module *must* be manually opened. It has no other dependencies.

Hardware Dependencies

- I/O port pin **PD5** on **PORTD** (for the RED LED)
- I/O port pin **PD6** on **PORTD** (for the GREEN LED)

Function List Summary

- `LED_open()` - Open and initialize the **LED** subsystem module.
- `LED_close()` - Close and release the **LED** subsystem module.
- `LED_set_pattern()` - Specify *bit pattern* for those LEDs that should be turned ON.
- `LED_clr_pattern()` - Specify *bit pattern* for those LEDs that should be turned OFF.
- `LED_tog_pattern()` - Specify *bit pattern* for those LEDs that should be *toggled*.
- `LED_set()` - Specify *which* LED should be turned ON.
- `LED_clr()` - Specify *which* LED should be turned OFF.
- `LED_toggle()` - Specify *which* LED should *toggle*.
- `LED_state()` - Specify the *state* of a specific LED to be turned ON, or OFF.

Function Reference

The `LED_open()` Function

Format:

`SUBSYS_OPENSTAT LED_open( void )`

Description:

Function acquires and initializes resources needed for operation of the **LED** subsystem module. You *must* call this function first with a successful 'open' before invoking any other function provided by this module.

Returns:

Returns a structure of type `SUBSYS_OPENSTAT` whose field entries indicate the status of the open request and the subsystem that resulted in an error (when, and *if* an error occurs). See *Chapter 1, Procedure for Opening Modules Before Use* for examples on how to handle the *open status*.

Example:

```
#include "capi324v221.h"

void CBOT_main( void )
{

  SUBSYS_OPENSTAT opstat;

  // Open the LED module.
  opstat = LED_open();

  if ( opstat.state == SUBSYS_OPEN )
  {

         // ... DO LED STUFF ...

  }// end if()

} // end CBOT_main()
```

The `LED_close()` Function

Format:

`void LED_close( void )`

Description:

Function deallocates and releases resources being used by the **LED** subsystem module. No other functions should be invoked once the subsystem module is closed.

The `LED_set_pattern()` Function

Format:

```
void LED_set_pattern( unsigned char LED_pattern )
```

Description:

Function allows you to set the *set bit pattern* for the LEDs. When a corresponding *bit* is set to **1** in the appropriate bit location it will turn the LED on. Nothing happens for bit positions whose values are **0**.

Input Arguments:

`LED_pattern` – This is an 8-bit value whose LEDs are mapped to the following *bit positions*:

```
bit0: n/a
bit1: n/a
bit2: n/a
bit3: n/a
bit4: n/a
bit5: Red   LED.
bit6: Green LED.
bit7: n/a
```

Example:

```
// Assume the LED module is already properly opened.

// Turn the RED LED ON.
LED_set_pattern( 0b00100000 );

// Turn the GREEN LED ON (the RED stays ON).
LED_set_pattern( 0b01000000 );

// Blah,  blah...
```

The `LED_clr_pattern()` Function

Format:

```
void LED_clr_pattern( unsigned char LED_pattern )
```

Description:

Function allows you to set the *clear bit pattern* for the LEDs. When a corresponding *bit* is set to **1** in the appropriate bit position, it will turn the LED off. Nothing happens for bit positions whose values are **0**.

(*Continued on next page*)

(*Continued from previous page*)

Input Arguments:

`LED_pattern` – This is an 8-bit value whose LEDs are mapped to the following *bit positions*:

```
bit0: n/a
bit1: n/a
bit2: n/a
bit3: n/a
bit4: n/a
bit5: Red   LED.
bit6: Green LED.
bit7: n/a
```

Example:

```
// Assume the LED module is already properly opened.

// Turn the RED LED ON.
LED_set_pattern( 0b00100000 );

// Turn the GREEN LED ON (the RED stays ON).
LED_set_pattern( 0b01000000 );

// Turn the RED LED OFF.
LED_clr_pattern( 0b00100000 );

// Turn the GREEN LED OFF.
LED_clr_pattern( 0b01000000 );

// Blah,  blah...
```

The `LED_tog_pattern()` Function

Format:

```
void LED_tog_pattern( unsigned char LED_pattern )
```

Description:

Function allows you to set the *toggle bit pattern* for the LEDs. When a corresponding *bit* is set to **`1`** in the appropriate bit position, it will toggle the state of the LED (e.g., if it's OFF, it will turn ON, and vice versa). Nothing happens for bit positions whose values are **`0`**.

Input Arguments:

`LED_pattern` – This is an 8-bit value whose LEDs are mapped to the following *bit positions*:

```
bit0: n/a
bit1: n/a
bit2: n/a
bit3: n/a
bit4: n/a
bit5: Red   LED.
bit6: Green LED.
bit7: n/a
```

Example:

```
// Assume the LED module is already properly opened.

unsigned int i;

// Toggle the RED LED ON and OFF a few times.
for ( i = 0; i < 10; i++ ) {

  // Toggle state.
  LED_tog_pattern( 0b00100000 );

  // Wait 1 second before the next iteration.
  //
  // NOTE:  See the 'Timer Service' (TMRSRVC) module documentation chapter
  //        for info on this function.
  //
  TMRSRVC_delay( TMR_SECS( 1 ) );

} // end for()
```

The `LED_set()` *Macro* Function

Format:

`LED_set( which )`

Description:

This macro-function allows you to turn ON a specific LED.

Input Arguments:

`which` – Must be one of the following:

`LED_Red` (or `LED0`)
`LED_Green` (or `LED1`)

(*Continued on next page*)

(*Continued from previous page*)

Example:

```
// Assume the LED module is already properly opened.

// Turn the RED LED on.
LED_set( LED_Red );

// Turn the GREEN LED on.
LED_set( LED_Green );
```

Note the helper macro is *less verbose* than using `LCD_set_pattern()`. However, `LCD_set_pattern()` allows you to control multiple LEDs at once where as the helper macros allow manipulation of one LED at a time.

The `LED_clr()` *Macro* Function

Format:

`LED_clr( which )`

Description:

This macro-function allows you to turn a specific LED OFF.

Input Arguments:

which – Must be one of the following:

`LED_Red`	(or `LED0`)
`LED_Green`	(or `LED1`)

Example:

```
// Assume the LED module is already properly opened.

// Turn the GREEN LED ON.
LED_set( LED_Red );

// Turn the GREEN LED OFF
LED_clr( LED_Green );
```

The `LED_toggle()` *Macro* Function

Format:

`LED_toggle( which )`

Description:

This macro-function allows you to *toggle* the state of a specific LED.

(*Continued on next page*)

Input Arguments:

which – Must be one of the following:

```
LED_Red      (or LED0)
LED_Green    (or LED1)
```

Example:

```
// Assume the LED module is already properly opened.

unsigned int i;

// Toggle the RED LED ON and OFF a few times.
for ( i = 0; i < 10; i++ ) {

  // Toggle state.
  LED_toggle( LED_Red );

  // Wait 1 second before the next iteration.
  //
  // NOTE:  See the 'Timer Service' (TMRSRVC) module documentation chapter
  //        for info on this function.
  //
  TMRSRVC_delay( TMR_SECS( 1 ) );

} // end for()
```

The `LED_state()` *Macro* Function

Format:

`LED_state( which, state )`

Description:

This macro function allows you to either turn a specific LED ON or OFF, depending on the *state* specified.

Input Arguments:

which – Must be one of the following:

```
LED_Red      (or LED0)
LED_Green    (or LED1)
```

state – Must be one of the following:

```
LED_ON
LED_OFF
```

Example:

```
// Assume the LED module is already properly opened.

// Turn the RED LED ON.
LED_state( LED_Red, LED_ON );

// Turn it OFF.
LED_state( LED_Red, LED_OFF );

// Turn the GREEN LED ON.
LED_state( LED_Green, LED_ON );

// and leave it on forever!....  Hahaha!
```

Chapter 7: The PSXC (Playstation Controller) Subsystem Module

This chapter covers the **PSXC** subsystem module and function reference. The **PSXC** provides services to access the Playstation 2-compliant controller such as reading button status, analog stick levels, etc.

Module at a Glance

Description

The **PSXC** subsystem module allows users to access the Playstation adapter present on the CEENBoT to read status information from any Playstation 2 compliant controller. When and *if* attached, the user can read status information from the push buttons, shoulder buttons, digital pad and analog sticks. Therefore, the **PSXC** module exists to facilitate these tasks.

Modular Dependencies

The **PSXC** module *must* be manually opened by the user. It has the following modular dependencies:

- **SPI** - The **PSXC** is an SPI device.

Hardware Dependencies

- *None* – other than what is acquired by the **SPI**, which this module depends on.

Function List Summary

- `PSXC_open()` - Open and initialize the **PSXC** module.
- `PSXC_close()` - Close and relinquish the **PSXC** module.
- `PSXC_read()` - Read PS2 controller data.
- `PSXC_get_center()` - Get *center* values from the *analog sticks* when left 'idle'.
- `PSXC_plinear_map()` - Fast *pseudo-linear* mapping function to map analog controller data to larger interval values. For example to map 0-128 to, perhaps, 0-1000.

Function Reference

The `PXSC_open()` Function

Format:

`SUBSYS_OPENSTAT PSXC_open( void )`

Description:

Function acquires and initializes resources needed for operation of the **PSXC**. You *must* call this function first with a successful 'open' before invoking any other function provided by this module.

Returns:

Returns a structure of type `SUBSYS_OPENSTAT` whose field entries indicate the status of the open request and the subsystem that resulted in an error (when, and *if* an error occurs). See *Chapter 1, Procedure for Opening Modules Before Use* for examples on how to handle the *open status*.

Example:

```
#include "capi324v221.h"

void CBOT_main( void )
{

  SUBSYS_OPENSTAT opstat;

  // Open the PSXC module.
  opstat = PSXC_open();

  if ( opstat.state == SUBSYS_OPEN )
  {

         // ... DO CONTROLLER STUFF ...

  }// end if()

} // end CBOT_main()
```

The `PSXC_close()` Function

Format:

`void PSXC_close( void )`

Description:

Function deallocates and releases resources being used by the **PSXC** subsystem module. No other functions should be invoked once the subsystem module is closed.

The `PXSC_read()` Function

Format:

```
BOOL PSXC_read( PSXC_STDATA *pStatus_data )
```

Description:

Use this function to read *state controller data* (push-buttons, digital pad, analog sticks, etc) when the PS2-type controller is attached and active. When invoking the function, you pass to it the *address of* a structure of type `PSXC_STDATA`. This structure contains fields (member data) which will be populated with corresponding controller data. In addition, the function returns a *boolean status* that can be used to determine if the *read* from the controller was successful. Users should take the habit to look at this return value as the `PSXC_read()` will populate the structure with *junk* if the read was not successful. This allows the user to decide what should be done once *illegitimate* data is intercepted and prevent this data from being passed along to other portions of your program.

Input Arguments:

`pStatus_data` – Pass to this argument the *address of* a structure of type `PSXC_STDATA`. When `PSXC_read()` completes, it will populate this structure with controller data. Note that the structure is populated with data whether the data read from the controller was actually successful or not, so *be careful!* Always check the returned status code to determine if the read was 'good'.

The structure of type `PSXC_STDATA` has the following form:

```
typedef struct PSXC_STDATA_TYPE {

    PSXC_TYPE data_type;      // Variable holds the type of data (analog or
                             // digital) or if it contains invalid data.

    unsigned char buttons0;   // Select/Start/DPAD and L3/R3 status.
    unsigned char buttons1;   // Shoulder buttons and right-hand side button
                             // status.

    // Holds analog data for the left analog joystick.  The values contained
    // range from -128 to 127, with ~0/1 being the IDEAL 'center'.
    struct {

        signed char up_down;
        signed char left_right;

    } left_joy;

    // Holds analog data for the right analog joystick. The values contained
    // range from -128 to 127, with ~0/1 being the IDEAL 'center'.
    struct {

        signed char up_down;
        signed char left_right;

    } right_joy;

} PSXC_STDATA;
```

Lets discuss the member data in the structure and how to interpret each field (*next page*):

(*Continued from previous page*)

`data_type` – This refers to the *type* of data returned from the controller. Possible values are:

`PSXC_INVALID` – Means the structure was populated with invalid data and should be ignored.
`PSXC_DIGITAL` – Means the structure was populated with data only applicable to *digital mode* operation of the controller (i.e., *analog controller* data should be ignored).
`PSXC_ANALOG` – Means the structure was populated with *analog data*.

`buttons0` – This is an 8-bit field, whose bit positions (being active LOW) are assigned as follows:

bit0: SELECT button.
bit1: Left Analog Joystick pushed (a.k.a L3) – *valid in analog mode only*.
bit2: Right Analog Joystick pushed (a.k.a R3) – *valid in analog mode only.*
bit3: START button.
bit4: Digital pad UP
bit5: Digital pad RIGHT
bit6: Digital pad DOWN
bit7: Digital pad LEFT

> **Note:** Careful! The bit positions in `buttons0` are ACTIVE LOW.

`buttons1` – This is an 8-bit field, whose bit positions (being active LOW) are assigned as follows:

bit0: L2 (on controller's *shoulder*)
bit1: R2 (on controller's *shoulder*)
bit2: L1 (on controller's *shoulder*)
bit3: R1 (on controller's *shoulder*)
bit4: TRI (*triangle* button)
bit5: CIR (*circle* button)
bit6: X ('X' button)
bit7:SQR (*square* button)

> **Note:** Careful! The bit positions in `buttons0` are ACTIVE LOW.

After this, there are two *substructures* or *inner structures* called `left_joy`, and `right_joy`, which refers to *analog joystick* data for the left and right joysticks respectively. Each of these substructures contain two fields for the respective analog joystick being:

`up_down` – Contains *analog controller data* (when in ANALOG MODE) which is proportional to the displacement of the analog stick of the controller. This particular field denotes displacement of the analog stick along the Y-axis (up/down motion). Its values range from (**-128** to **+127**), with **-1** to **1** being the *ideal center*, **-128** meaning all the way DOWN, and **+127** meaning all the way UP.

(*Continued on next page*)

(*Continued from previous page*)

`left_right` – Contains *analog controller data* (when in ANALOG MODE) which is proportional to the displacement of the analog stick of the controller. This particular field denotes the displacement of the analog stick along the X-axis (left/right motion). Its values range from (**-128** to **+127**), with **-1** to **1** being the *ideal center*, **-128** meaning all the way LEFT, and **+127** meaning all the way RIGHT.

Returns:

The function returns a status of type BOOL (*boolean*) to indicate if the 'read' was good or not. If `TRUE`, the data can be trusted – otherwise, it should be discarded. Reasons for invalid data could be low-batteries, error in communication, controller is of incompatible type (manufacturers vary), controller not fully connected, or perhaps not fully awake, etc.

Example:

Let us look at a simple example:

For this first example, assume that the following function exists (it *doesn't*):

```
motors_move( signed char left_speed, signed char right_speed )
```

Where `left_speed` and `right_speed` just happen to also take the values ranging from **-128** (full reverse) to **+127** (full forward), with 0 being *no motion*, which is GREAT for our fake scenario because **PSXC** module returns analog data in this exact same range – how wonderful! So, the goal here is to use the analog sticks to control the motors using our *fake function* – here's how you do it:

```
// Assume the PSXC module has been properly opened
// via 'PSXC_open()'.

// We want to continuously read the controller data,
// and as long as this data is good, we'll use it to control
// the motors with the fake function 'motors_move()'.

// Declare structure to store controller data when polled.
PSXC_STDATA    psxc_data;

// Declare storage to store our analog joystick data.
signed char L_up_down;
signed char R_up_down;
```

(*Continued on next page*)

(*Continued from previous page*)

```
// Enter the infinite loop, but only bother if the data is good.
while( 1 )
{

  // ...butonly bother with it if the data is good.
  if( PSXC_read( &psxc_data ) == TRUE )
  {

        // Only take action if VALID ANALOG data was read.
        if ( psxc_data.data_type == PSXC_ANALOG )
        {

              // Read only the 'up/down' parameters for left and right sticks.
              L_up_down = psxc_data.left_joy.up_down;
              R_up_down = psxc_data.right_joy.up_down;

              // Pass it to our 'fake' function.
              motors_move( L_up_down, R_up_down );

        } // end if()

        else
        {

              // Otherwise, as a safety mechanism pass safe values to
              // our 'fake' function (just STOP the motors).
              motors_move( 0, 0 );

        } // end else.

        // Regardless of whether the data is ANALOG or DIGITAL,
        // read one of the 'buttons' fields so we can get out of
        // this loop when the user presses the [START] button.
        if ( !( psxc_data.buttons0 & STRT_BIT ) )

              break; // Break out of the 'while()' loop.

  } // end if()

} // end while()

// Blah, Blah..., etc.
```

Note, in the example above the part where we check to see if the *START* button is been pressed. Here we do a *bit-wise AND* and then *invert* – recall as already stated – button data is ACTIVE LOW! The 'STRT_BIT' is a macro constant predefined in the header file. As a matter or fact, the following macro-constants can be used to test bit positions (*see next page*):

Bit positions that can be 'bit-wise' tested with **buttons0** in the PSXC_STDATA structure:

```
#define SLCT_BIT        0x01
#define JOYR_BIT        0x02        /* Analog mode only */
#define JOYL_BIT        0x04        /* Analog mode only */

#define L3_BIT          JOYR_BIT    /* Alternate names (Analog mode only) */
#define R3_BIT          JOYL_BIT    /* Alternate names (Analog mode only) */

#define STRT_BIT        0x08
#define DPAD_UP_BIT     0x10    /* Digital pad UP    */
#define DPAD_RT_BIT     0x20    /* Digital pad RIGHT */
#define DPAD_DN_BIT     0x40    /* Digital pad DOWN  */
#define DPAD_LT_BIT     0x80    /* Digital pad LEFT  */
```

Bit positions that can be 'bit-wise' tested with **buttons1** in the PSXC_STDATA structure:

```
#define L2_BIT          0x01
#define R2_BIT          0x02
#define L1_BIT          0x04
#define R1_BIT          0x08
#define TRI_BIT         0x10
#define CIR_BIT         0x20
#define X_BIT           0x40
#define SQR_BIT         0x80
```

Here's one last example – this one implements the *bit definitions* given above:

```
// Assume the PSXC module has been properly opened
// via 'PSXC_open()'.

// We want to continuously read the controller data,
// and as long as this data is good, we'll use it to
// determine if the CIRCLE or TRIANGLE buttons are pressed.


// Declare structure to store controller data when polled.
PSXC_STDATA      psxc_data;


// Enter the infinite loop, but only bother if the data is good.
while( 1 )
{

  // ...butonly bother with it if the data is good.
  if( PSXC_read( &psxc_data ) == TRUE )
  {

         // Check if CIRCLE button pressed.
         if ( !( psxc_data.buttons1 & CIR_BIT ) )
         {

                // Do whatever you need to do.
                foo();

         } // end if()
```

(*Continued on next page*)

(*Continued from previous page*)

```
            // Check if TRIANGLE button pressed.
            if ( !( psxc_data.buttons1 & TRI_BIT ) )
            {

                // Do whatever you need to do.
                bar();

            } // end if()

            // Regardless of whether the data is ANALOG or DIGITAL,
            // read one of the 'buttons' fields so we can get out of
            // this loop when the user presses the [START] button.
            if ( !( psxc_data.buttons0 & STRT_BIT ) )

                break;  // Break out of the 'while()' loop.

        } // end if()

    } // end while()
```

The `PSXC_get_center()` Function

Format:

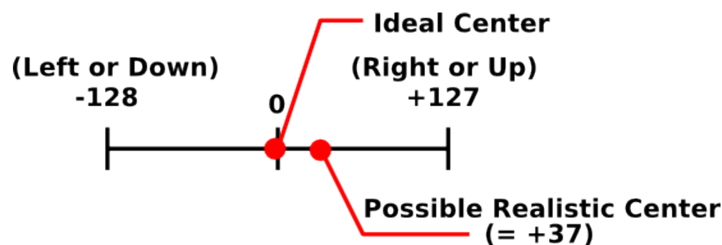
```
BOOL PSXC_get_center( PSXC_CENTER *pCenter_data )
```

Description and Motivation:

Every time you call `PSXC_read()` to read controller data/status, the **PSXC** module internally stores the very first *valid* analog data it reads from the controller. This data is *assumed* to be the center values of the analog sticks when they're left to sit idle at the center. If on the course of reading controller data (after the internal *center* values are already stored) the data is suddenly invalid (e.g. you turn the controller OFF, and then back ON), the **PSXC** module will store *new* 'center data' values once it catches the very first valid analog data the next time around.

If the user wishes, he/she can then ask "*what is the latest valid center values for the analog sticks?*" by requesting this info by calling `PSXC_get_center()`. The user can then use this information for calibration purposes to compensate for non-ideal center values returned by the controller due to variability among controller manufacturers.

In an perfect world, **PSXC** controller data would always return values between `-1` and `1` (or even `0`) for left/right and up/down axes when the sticks are left idle at the center. However, this is rarely the case and the user should NEVER assume that analog stick controller data will always fall between a predetermined range. This potential scenario is visually illustrated below:



(*Continued from previous page*)

All this function does is inform you of what the **PSXC** module 'thinks' the center values are when it intercepted valid analog data for the first time. What you do with this information and how you choose to compensate for potential "off-center" errors is up to you.

One solution is to use this function in conjunction with the `PSXC_plinear_map()` function (discussed next). The example shown for *that* function might give you some ideas.

Input Arguments:

`pCenter_data` – You must pass to this argument the *address of* a structure of type `PSXC_CENTER`. The internal structure of this data type is given below:

```
typedef struct PSXC_CENTER_TYPE {

    struct {

        signed char up_down;
        signed char left_right;

    } left_joy;

    struct {

        signed char up_down;
        signed char left_right;

    } right_joy;

} PSXC_CENTER;
```

`PSXC_get_center()` will then populate this structure with appropriate values.

Returns:

Function returns a value of type `BOOL` (*boolean*). It will return `TRUE` if the structure was populated with *valid* data. It will return `FALSE` if the structure was populated with *invalid* data, for example, if it still hasn't had a chance to store valid 'center data' because the user hasn't issued a `PSXC_read()`.

Example:

The following example shows how to properly use `PSXC_get_center()`. It, however, does NOT show how you would use this data to compensate for "off-center error" – see the `PSXC_plinear_map()` for one possible way you might accomplish this. This example shows what you would probably do, for example, at the beginning of your program, where you need to calibrate and correct or compensate for "off-center" errors. The example assumes a 'fake function' called calibrate() exists, which takes these 'center values' and does the rest.

```
// Assume the PSXC module has been properly opened
// via 'PSXC_open()'.

PSXC_CENTER center_values;  // Structure to hold CENTER values.
PSXC_STDATA controller_data;   // Structure to hold CONTROLLER data.

// First step is to wait until the user connects, activates, or
// enables the controller in ANALOG MODE.
while( PSXC_read( &controller_data ) == PSXC_INVALID );
```

(*Continued on next page*)

(*Continued from previous page*)

```
// If we get past the while loop we have at least one successful
// polling of data from the controller.  This means PSXC should
// now have stored center data.  So lets get center data and
// store it for calibration (however you choose to do it).
if ( PSXC_get_center( &center_values ) == TRUE )
{

  // Perhaps pass this to a 'fake' function call 'calibrate()'
  // to do what is necessary.
  calibrate( &center_values );

} // end if()

// ... go about your business and read controller data …
```

The `PSXC_plinear_map()` Function

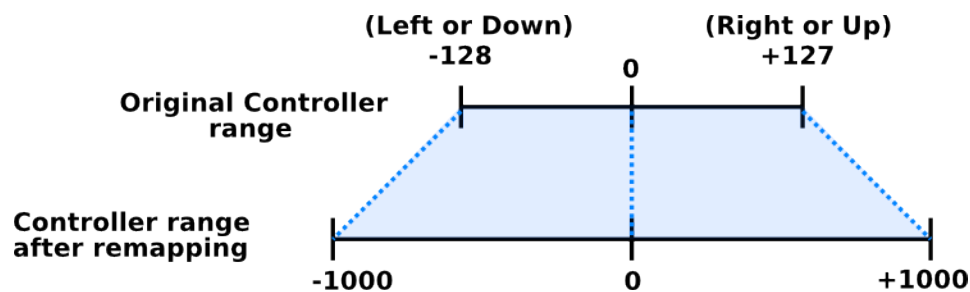
Format:

```
signed short int PSXC_plinear_map( signed char analog_val,
                                   signed char min_in_val,
                                   signed char max_in_val,
                                   signed short int max_out_val )
```

Description and Motivation:

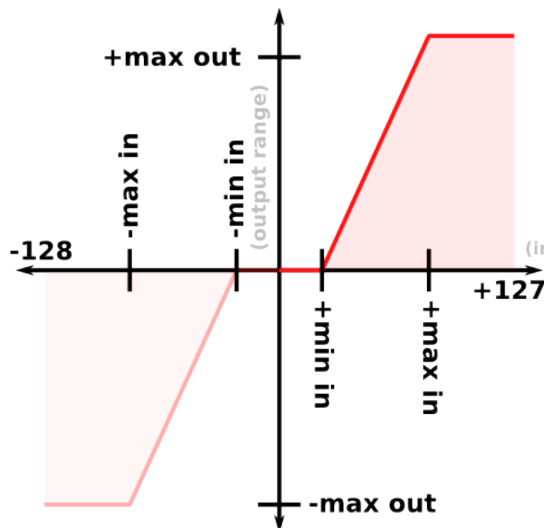
When reading the *analog values* for the *analog sticks* via `PSXC_read()`, the values for up/down and left/right axes range between **`-128`** to **`+127`**, with **`0`** being the *ideal* center position between the two extremes. As long as the *device* or *object* you're trying to control also takes on values of this exact same range you'll have no problem.

However, what if we want to use the controller values from the analog sticks from **`-128`** to **`+127`** to control a device or object that takes values from **`-1000`** to **`+1000`**? We do so with `PSXC_plinear_map()` function! So, as you might guess, we can use this function to map the default interval from the original interval of **`-128`** to **`+127`** range to something greater (or smaller) than that. This principle is illustrated below:



Note that the *remapping* can only be done symmetrically – that is, zero maps to zero (always), and whatever the mapping is for the *positive interval*, the same is duplicated for the *negative interval*. In fact, remapping can ONLY be specified for the *positive interval* and the *negative interval* is duplicated accordingly.

An additional benefit of the `PSXC_plinear_map()` function is that it can be used as a solution to the “off-center” issue described with the `PSXC_get_center()` function previously discussed by creating a *dead-zone* about the 'zero position' as shown in the next figure (*see next page*).



In the above figure the *x-axis* represents the range of values that can be obtained from the *analog sticks* by way of `PSXC_read()` function. The *y-axis* represents the remapped values via `PSXC_plinear_map()`. Note there are three *key* parameters to consider here. For the input values we have *min in* and *max in*. Input values below *min in* will be clipped to zero. This creates a *dead-zone* between -*min in* and +*min in*. In a similar manner, values greater than +*max in* will be mapped to the specified +*max out* – this is the maximum expected output value.

Input Arguments:

`analog_val` – You pass to this argument the *raw analog controller* value from one of the analog sticks up/down or left/right (obtained via `PSXC_read()`), which will range from `-128` to `+127` as already stated.

`min_in_val` – This is the minimum *input value* from the controller such that values below it will be mapped to zero.

`max_in_val` – This is the maximum *input value* from the controller such that values above it will be mapped to the specified *maximum output value* (given by '`max_out_val`' below).

`max_out_val` – This is the maximum permissible *output value* that remapping is allowed to generate. No values above this will be returned by the function when called to remap.

> **Note:** Remember the *remapping* is done *symmetrically*. You ONLY specify the positive 'quadrant' as shown in the above figure, and the negative 'quadrant' is replicated in a similar way.

> **Note:** The following relation *must* hold, regarding the following parameters:
>
> `0 <= min_in_val < max_in_val <= +127`
>
> `-128 <= analog_val <= +127`
>
> `0 <= max_out_val <= 32767`

Returns:

The function returns a value of type `signed short int` corresponding to the *newly mapped* value based on the mapping parameters supplied to the input arguments of the function.

Examples:

In this example we're going to do two things, we're going to get the *center values* once they're captured by the **PSXC** module and then we'll use that to determine how big our *dead zone* should be (the range of values from the controller *about* zero that will be mapped *to* zero – that is, the controller's *ideal* center) in addition to mapping the analog sticks range from **-128** to **+127** to **-1000** to **+1000** and feed this information to a *fake function* called:

```
motors_move( unsigned short int left_speed, unsigned short int right_speed )
```

whose parameters can range between **-1000** to **+1000** necessitating a need for the *remapping* we're about to do. So here goes:

```
// Assume the PSXC module has been properly opened
// via 'PSXC_open()'.

// Suppose we declare a function called 'process_controller()' (with prototype
// declare elsewhere) which takes care of reading the controller analog sticks
//  values and using this result (after remapping) to control the motors via
// 'motors_move()'.

void process_controller( void )
{

    PSXC_STDATA controller;     // For storing controller state data.
    PSXC_CENTER curr_center;    // For storing current 'center' data.

    static BOOL got_center = FALSE;

    // Left analog stick.
    signed short int aleft_LR;  // Left/Right value.
    signed short int aleft_UD;  // Up/Down    value.

    // Right analog stick.
    signed short int aright_LR; // Left/Right value.
    signed short int aright_UD; // Up/Down    value.

    // Read controller and process only if data is valid.
    if ( PSXC_read( &controller ) == TRUE )
    {

         // Get center data if we don't have it, or if previously acquired
         // 'center values' have been INVALIDATED.
         if ( got_center == FALSE )
         {
                // Get new 'center values' and set the state variable 'got_center'
                // to indicate that we got it so we don't do this again the next
                // time around UNTIL an INVALIDATION occurs for whatever reason.
                got_center = PSXC_get_center( &curr_center );

         } // end if()
```

```
        // Then, do remapping ONLY if we have successfully received
        // 'center values' AND if we're in ANALOG MODE.
        if ( ( got_center == TRUE ) && ( controller.data_type == PSXC_ANALOG ) )
        {

        // Process the mapping.
        aleft_LR = PSXC_plinear_map( controller.left_joy.left_right,

                  ( curr_center.left_joy.left_right + PADDING ), 127, 1000 );

        aleft_UD = PSXC_plinear_map( controller.left_joy.up_down,

                  ( curr_center.left_joy.up_down + PADDING ), 127, 1000 );

        aright_LR = PSXC_plinear_map( controller.right_joy.left_right,

                  ( curr_center.right_joy.left_right + PADDING ), 127, 1000 );

        aright_UD = PSXC_plinear_map( controller.right_joy.up_down,

                  ( curr_center.right_joy.up_down + PADDING ), 127, 1000 );

        } // end if()

        // Otherwise, set to safe values, since it appears our data
       // cannot be trusted.
        else
        {

               aleft_LR = 0;
               aleft_UD = 0;

               aright_LR = 0;
               aright_UD = 0;

        } // end else.

    } // end if()

    else
    {

        // If we can't even read data from the controller, we still have
        // to pass 'safe' values to our 'motors_move()' function, so let's
        // make sure that happens by setting the analog sticks values to zero.
        aleft_LR = 0;
        aleft_UD = 0;

        aright_LR = 0;
        aright_UD = 0;
```

```
            // Invalidate CENTER data -- this will force the retrieval of
            // new 'center values' to recompute the 'dead-zone'
            // once VALID data does arrive -- whenver that happens, IF ever.
            got_center = FALSE;

        } // end else.

    // Finally send these values to our fake function to get the
    // motors moving.  We only use the data pertaining to the up/down
    // axes for the analog sticks.  The left/right axes are read JUST
    // TO SHOW, but we don't need that information in this case.
    motors_move( aleft_UD, aright_UD );

} // end proceess_controller()
```

That's just the function definition. You would then put this function, perhaps in a *while* loop, where it repeatedly reads controller data so that action can always be taken, like so:

```
// Assume 'PSXC' module is already OPEN.

// ... code here...

while( 1 ) {

  // Read controller data and move motors with it.
  process_controller();

  // Do whatever else you need to do...

} // end while()

// etc, etc...
```

Let's summarize what is going on in the `process_controller()` function.

We start by declaring storage... two structures, one called `controller`, to store controller data once *polled*, and the other called `curr_center` to store *center values* from the analog sticks.

Next comes `got_center`. This is a *boolean* to keep track if we have already obtained new 'center values' or not. We don't want to continuously read center values on each iteration – that would be a waste... we only need new center values if controller data becomes *invalidated* due to a disconnection or what have you – we use this variable to control *when* this is allowed to happen.

Next comes storage for our analog values *after* they have been remapped to the new range. Recall we want to map **-128** to **+127** to the range **-1000** to **+1000**. Here we create storage for both axes for the left analog joystick and right analog joystick (*up/down* and *left/right*).

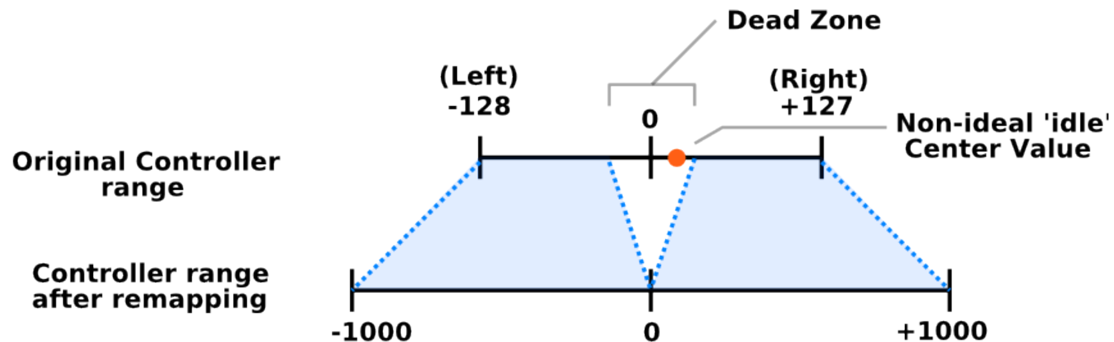
In the first *if* block, we read controller data, and store the result in the controller structure and at the same time we check if the structure is populated with valid data. If not, nothing will happen.

Assume the read is good, we move into the first inner *if* block and check if we have 'center values'. If this is the first time into the function the answer to that question is “Not Yet...”, so inside, `PSXC_get_center()` is called, and if the data is good, the variable `got_center` will be updated to indicate “so far so good.”

If the `got_center` variable is `TRUE` (meaning acquiring center data was successful), then in the next *if* block we proceed to *remapping*, but only if the controller has received data pertaining to ANALOG MODE. Let's look at one of the functions:

```
aleft_LR = PSXC_plinear_map( controller.left_joy.left_right,
                             ( curr_center.left_joy.left_right + PADDING ), 127, 1000 );
```

Here we're telling `PSXC_plinear_map()` to map from the whatever the 'idle center value' is (with a little *padding* added) up to **`+127`**, to the range **`0`** to **`+1000`** (on the positive side – remember the *negative* side follows suit). The figure below shows what `PSXC_plinear_map()` will do for this particular line of code:



Effect of remapping for Analog Stick Left/Right Axis

The *red dot* shows where the 'idle center' value *might* be. Therefore, this has to be part of our *dead-zone*. We don't want the edge of the dead-zone to coincide exactly with that value, that's why we add a little extra 'padding' as shown in the code.

In any case the same process is repeated for all other analog controller axes. After that, the *remapped* values are properly passed to `motors_move()` for action to be taken.

How much 'safety' and how much detail you want to implement in your routines when reading controller data is up to you, and of course is specific to the scenario at hand. Just know that facilities exist to simply this process.

Chapter 8: The SPI Subsystem Module

This chapter introduces you to SPI functional services provided by the CEENBoT-API.

Module at a Glance

Description

The **SPI** subsystem module provides functional services for making use of the *Serial Peripheral Interface* subsystem made available by the MCU. Functions allow you to control features of this device. Presently only a limited feature set is provided but additional features will be added as development continues on the API.

Modular Dependencies

The **SPI** module has no modular dependencies. This service is *open by default*.

Hardware Dependencies

- I/O port pin **PB5** on **PORTB** (for SPI **MOSI**)
- I/O port pin **PB6** on **PORTB** (for SPI **MISO**)
- I/O port pin **PB7** on **PORTB** (for **SCLK**)
- I/O port pin **PB0** on **PORTB** (for SPI slave addressing **SS0**)
- I/O port pin **PB1** on **PORTB** (for SPI slave addressing **SS1**)
- I/O port pin **PB2** on **PORTB** (for SPI slave addressing **SS2**)

The three *slave addressing* lines are fed to a 3-to-8 decoder, where each select line from the decoder's output is used to select one of 8 possible addressable slave devices. Now, not all of these SPI slave device addresses are available – the SPI slave assignment in the '324 v2.21 platform of the CEENBoT is as follows:

SPI_DEV_ADDR0 – Assigned to LCD device
SPI_DEV_ADDR1 – Assigned to PSXC controller
SPI_DEV_ADDR2 – Assigned to secondary MCU (the **ATtiny**)
SPI_DEV_ADDR3 – Available via on-board *expansion header* (**J3 – pin 9**)
SPI_DEV_ADDR4 – Available via on-board *expansion header* (**J3 – pin 10; Also J5 – via DB9**)
SPI_DEV_ADDR5 – Available via on-board *expansion header* (**J3 – pin 11; Also J6 – via DB9**)
SPI_DEV_ADDR6 – *Reserved*.
SPI_DEV_ADDR7 – *Reserved* (used as SPI 'message separator' or *null* addressing to initiate a 'new' SPI message stream)

Note that only SPI slave addresses 3, 4, and 5 are available for *your* use.

In addition to the select lines being exposed via the on-board expansion header on connector **J3**, the SPI slave address lines 4 and 5 as well as the corresponding primary SPI communication lines (**MOSI**, **MISO**, and **SCLK**) are also available via the two on-board 9-pin serial connectors **J5** and **J6**.

> **Note:** You must consult the *schematic* for the '324 v2.21 platform diagram to see what the *pin assignment* is for connecting your SPI devices via the DB9 connectors.

> **Note:** The DB9 pins assignments are also shared with the I^{2}C (TWI) device lines. See the chapter on I^{2}C (TWI) for details.

Function List Summary

- `SPI_open()` - Open and initialize acquired resources for the **SPI** subsystem module.
- `SPI_close()` - Close and release acquired resources for the **SPI** subsystem module.
- `SPI_set_slave_addr()` - Sets the SPI slave address for subsequent data transmissions.
- `SPI_transmit()` - Function used to transmit data (in MASTER mode) to the selected slave device.
- `SPI_receive()` - Function used to receive data (in MASTER mode) from the selected slave device.
- `SPI_set_config_func()` - Function used to register your *configuration function* for a given SPI slave address.

Function Reference

The `SPI_open()` Function

Format:

```
SUBSYS_OPENSTAT SPI_open( void )
```

Description:

Function acquires and initializes resources needed for operation of the **SPI**. You *must* call this function with a successful 'open' before invoking any other function provided by this module.

Returns:

Returns a structure of type SUBSYS_OPENSTAT whose field entries indicate the status of the open request and the subsystem that resulted in an error (when, and *if* an error occurs). See *Chapter 1, Procedure for Opening Modules Before Use* for examples on how to handle the *open status*.

> **Note:** Recall the **SPI** subsystem module is open by default.

The `SPI_close()` Function

Format:

```
void SPI_close( void )
```

Description:

Function deallocates and releases resources being used by the **SPI** subsystem module. No other functions should be invoked once the subsystem module is closed.

> **Note:** You shouldn't attempt to *close* the **SPI** module – it is a *critical* component of the API.

The `SPI_set_slave_addr()` Function

Format:

```
void SPI_set_slave_addr( SPI_SSADDR slaveSelectAddr )
```

Description:

This function can be used to *select* an *SPI slave device*.

Input Arguments:

`slaveSelectAddr` – Must be one of the following enumerated constants:

> `SPI_DEV_ADDR0` (or `SPI_ADDR_LCD`)
> `SPI_DEV_ADDR1` (or `SPI_ADDR_PSXC`)
> `SPI_DEV_ADDR2` (or `SPI_ADDR_ATTINY0`)
> `SPI_DEV_ADRR3` (or `SPI_ADDR_SMARTDEV0`)
> `SPI_DEV_ADDR4` (or `SPI_ADDR_SMARTDEV1`)
> `SPI_DEV_ADDR5` (or `SPI_ADDR_SMARTDEV2`)
> `SPI_DEV_ADDR6`
> `SPI_DEV_ADDR7` (or `SPI_ADDR_NA`)
>
> Note the `SPI_DEV_ADDRn` is the generic SPI slave address. The alternate names given in parenthesis are more descriptive, but you can use either one.

Example:

Both the `SPI_transmit()` and `SPI_receive()` functions (discussed in this chapter) allow you to specify the address to write or receive from. However, there are times where the SPI address must be set with no data to send or receive, or separate various SPI message streams by addressing the *null* device which is what the SPI slave address 7 is for – this is illustrated in the following example:

```
// Assume the 'SPI' subsystem module has bee properly opened.


// Suppose your SPI device always expects 3 bytes and no more.  To send
// and additional 3 bytes you have to restart a new 'message stream', which
// is marked by re-selecting the same SPI again.


// Suppose the SPI slave address is currently sitting on Device Address 7 (the
// NULL device).  We select our device and transmit data at the same time as
// follows:

// NOTE: The 'data' values are just random.

SPI_transmit( SPI_DEV_ADDR4, 0x78 );
SPI_transmit( SPI_DEV_ADDR4, 0xF0 );
SPI_transmit( SPI_DEV_ADDR4, 0x00 );
```

.(*Continued on next page*)

(*Continued from previous page*)

```
// We deselect the device by selecting the NULL device address
// so we can start a new 3-byte sequence message stream.

SPI_set_slave_addr( SPI_ADDR_NA );// I'm using the 'alternate name'
                                  // to 'SPI_DEV_ADDR7'.

// Repeat the process (select the SPI device and transmit)...

SPI_transmit( SPI_DEV_ADDR4, 0x78 );
SPI_transmit( SPI_DEV_ADDR4, 0xF0 );
SPI_transmit( SPI_DEV_ADDR4, 0x00 );

// We're done.  Deselect our SPI device.
SPI_set_slave_addr( SPI_ADDR_NA );
```

Note that you should always select `SPI_ADDR_NA` (or `SPI_DEV_ADDR7`) when no SPI is selected. Also, note how `SPI_set_slave_addr()` comes in handy for changing the address selection when no transmission or reception of data is required.

The `SPI_transmit()` Function

Format:

```
void SPI_transmit( SPI_SSADDR slaveSelectAddr, unsigned char data )
```

Description:

Use this function to transmit data (as MASTER) to the specified SPI slave device.

Input Arguments:

`slaveSelectAddr` – Must be one of the following enumerated constants:

> `SPI_DEV_ADDR0` (or `SPI_ADDR_LCD`)
> `SPI_DEV_ADDR1` (or `SPI_ADDR_PSXC`)
> `SPI_DEV_ADDR2` (or `SPI_ADDR_ATTINY0`)
> `SPI_DEV_ADRR3` (or `SPI_ADDR_SMARTDEV0`)
> `SPI_DEV_ADDR4` (or `SPI_ADDR_SMARTDEV1`)
> `SPI_DEV_ADDR5` (or `SPI_ADDR_SMARTDEV2`)
> `SPI_DEV_ADDR6`
> `SPI_DEV_ADDR7` (or `SPI_ADDR_NA`)
>
> Note the `SPI_DEV_ADDR`*n* is the generic SPI slave address. The alternate names given in parenthesis are more descriptive, but you can use either one.

`data` – The data *byte* to send.

Example:

See the example given for the function `SPI_set_slave_addr()` (above).

The `SPI_receive()` Function

Format:

```
char SPI_receive( SPI_SSADDR slaveSelectAddr, unsigned char data )
```

Description:

Use this function to receive data (as MASTER) from the specified SPI slave device.

Input Arguments:

`slaveSelectAddr` – Must be one of the following enumerated constants:

> `SPI_DEV_ADDR0` (or `SPI_ADDR_LCD`)
> `SPI_DEV_ADDR1` (or `SPI_ADDR_PSXC`)
> `SPI_DEV_ADDR2` (or `SPI_ADDR_ATTINY0`)
> `SPI_DEV_ADRR3` (or `SPI_ADDR_SMARTDEV0`)
> `SPI_DEV_ADDR4` (or `SPI_ADDR_SMARTDEV1`)
> `SPI_DEV_ADDR5` (or `SPI_ADDR_SMARTDEV2`)
> `SPI_DEV_ADDR6`
> `SPI_DEV_ADDR7` (or `SPI_ADDR_NA`)
>
> Note the `SPI_DEV_ADDR`*n* is the generic SPI slave address. The alternate names given in parenthesis are more descriptive, but you can use either one.

`data` – Sometimes you need to *transmit* data as you receive! You can use this parameter to specify the data to send as you receive data. Otherwise you can just set to `0` (or use the predefined macro-constant `SPI_NULL_DATA`, which also stands for '`0`').

Returns:

The function returns a data *byte* of type `char`. This value represents the *received data* from the selected slave.

Example:

```
// Assume the 'SPI' subsystem module has bee properly opened.

unsigned int i = 0;
unsigned char spi_data[ 3 ] = { 0 };   // Storage for our data.

// Suppose the SPI slave address is currently sitting on Device Address 7.
// Suppose we need to read three consecutive bytes from the selected device.

for ( i = 0; i < 3; i++ ) {
                                // NOTE: 'SPI_NULL_DATA' is the same as '0'.
  spi_data[ i ] = SPI_receive( SPI_DEV_ADDR5, SPI_NULL_DATA );

} // end for()

// We're done.  Deselect our SPI device.
SPI_set_slave_addr( SPI_ADDR_NA );

// ... Continue doing whatever...
```

The `SPI_set_config_func()` Function

Format:

```
void SPI_set_config_func( SPI_SSADDR ssAddr, SPI_CONFIGFUNC pConfigFunction )
```

Description and Motivation:

As you may have gathered by now, the CEENBoT platform contains various SPI devices. However, the configuration used to talk to each of these SPI devices is *not* similar. Some devices can work (transmit and receive) at higher clock speeds than others and may have a need for the SPI subsystem (on the MCU) to be configured with particular clock rate, clock polarity and active clock edge sequences in which the SPI will operate.

Therefore, before communicating with your custom SPI device (provided you attach one via the free addressable SPI slave device lines), *you* must configure the SPI subsystem on the MCU to successfully do so before talking to your SPI device.

Therefore, to allow your SPI to work in a *heterogeneous* environment, you have to write what is called an *SPI configuration routine*, which you do so by using a special macro called `SPI_CFGR()`. Your configuration routine sets up the registers to configure the SPI subsystem on the MCU. How to do this is not within the scope of this document so you must refer to the MCU's datasheet in order to do so. However, as an example, here's what the SPI configuration routine looks like for the **`PSXC`** module to access the Playstation 2 controller via SPI.

```
SPI_CFGR( SPI_PSXC_config )
{

    // Setup the SPCR register (see data-sheet of ATmega324 for details).
    SPCR = ( 1 << SPE  ) | ( 1 << DORD ) | ( 1 << MSTR ) | ( 1 << CPOL ) |
           ( 1 << CPHA ) | ( 1 << SPR1 ) | ( 1 << SPR0 );

    // Setup clock divider (Fosc/128).
    SPSR = 0;

    // Wait a bit.
    DELAY_us( 100 );

} // end SPI_PSXC_confg()
```

Basically every time the PS2 controller is *addressed* on the SPI (such as when you call `PSXC_read()`), the SPI subsystem in the MCU is configured by the above routine. This is done only once during initial addressing. If you access another SPI (say the LCD, which is also an SPI device), then the configuration routine for the LCD (which is different) will be run once to set up the SPI subsystem on the MCU to communicate with the LCD.

Therefore, *you* have to write your own SPI configuration routine to configure the SPI the way *you* want it in order to talk to your SPI, provided you have an SPI device attached to one of the freely available SPI slave address lines. Then, you would use `SPI_set_config_func()` function to *register* your configuration routine with the **`SPI`** subsystem module of the CEENBoT-API. Check out the 'Example' section below to see how this is done.

Input Arguments:

`ssAddr` – You must specify one of the following enumerated constants to specify the slave address that your custom configuration routine will be associated with.

`SPI_DEV_ADDR0` (or `SPI_ADDR_LCD`)
`SPI_DEV_ADDR1` (or `SPI_ADDR_PSXC`)
`SPI_DEV_ADDR2` (or `SPI_ADDR_ATTINY0`)
`SPI_DEV_ADRR3` (or `SPI_ADDR_SMARTDEV0`)
`SPI_DEV_ADDR4` (or `SPI_ADDR_SMARTDEV1`)
`SPI_DEV_ADDR5` (or `SPI_ADDR_SMARTDEV2`)
`SPI_DEV_ADDR6`
`SPI_DEV_ADDR7` (or `SPI_ADDR_NA`)

Note the `SPI_DEV_ADDR`*n* is the generic SPI slave address. The alternate names given in parenthesis are more descriptive, but you can use either one.

`pConfigFunction` – You must specify for this argument the *name* of your SPI configuration routine declared using the `SPI_CFGR(` *name* `)` macro. You declare an SPI configuration function as follows:

```
// Assume the 'SPI' subsystem module has bee properly opened.

// This is the 'prototype' -- just like any function, you NEED a prototype, and
// you can put this in your custom 'header' file if you have one.
SPI_CFGR( my_spi_device );


// This is your function 'definition'.  Like any other function, you define it
// outside of 'CBOT_main()'.
SPI_CFGR( my_spi_device )
{

  // Your ATmega324-specific instructions here.

} // end my_spi_device()
```

Then, you pass `my_spi_device` as an argument to this parameter. See the 'Example' below for a more comprehensive example.

Example:

Suppose you have an SPI device attached to one of the DB9 connectors on the CEENBoT, say, on SPI slave address 4. You would then need to declare your own custom SPI configuration routine to associate with this slave address and then register it. Obviously, you would do this as your program is just starting and initializing 'stuff'. This is a more comprehensive example (*see next page*):

```
#include "capi324v221.h"

// ==================== prototypes ======================
SPI_CFGR( my_spi_device );

// ==================== functions =======================
SPI_CFGR( my_spi_device )
{

  // ... ATmega324-SPI-specific stuff goes here...
  //       (see the 'datasheet' for this!)

}

// ====================== CBOT main ======================
void CBOT_main( void )
{

  // ... initialize stuff,  blah, blah...

  // Register our SPI configuration routine and associate it
  // with slave address 4.
  SPI_set_config_func( SPI_DEV_ADDR4, my_spi_device );

  // Now we can start talking to our device.
  SPI_transmit( SPI_DEV_ADDR4,  0x26 );
  SPI_transmit( SPI_DEV_ADDR4,  0x77 );

  // ... etc, etc...

  // Done -- select the NULL device.
  SPI_set_slave_addr( SPI_ADDR_NA );

  // ... now do something else...

  while( 1 ); // Never leave.

} // end CBOT_main()
```

Chapter 9: The SPKR (Speaker) Subsystem Module

> This chapter introduces you to SPKR functional services provided by the CEENBoT-API. This module provides services for generating audible tones on the CEENBoT, via the on-board speaker.

Module at a Glance

Description

The **SPKR** subsystem module exposes functional services that allow the user to generate a variety of audible tones via the CEENBoT's on-board speaker. There are two modes of operation in which the **SPKR** module can operate in, both of which can be active at the same time. These modes are:

- *Tone* mode
- *Beep* mode

The differences and peculiarities of each mode are discussed next.

About the *Tone* Mode

To use the **SPKR** subsystem module in *tone* mode, you have to open the **SPKR** subsystem module via `SPKR_open()` while specifying the mode to be that of *tone* as shown below:

```
SPKR_open( SPKR_TONE_MODE )
```

Tone mode affords you the following advantages:

- You can specify frequencies with *decimal* precision (e.g., **875.2 Hz**). Mind you, it may not be very *accurate,* but it allows for tones corresponding to frequencies of musical notes to be easily derived sufficiently close to the true frequency of a musical note.

- You can generate tones with frequencies up to **1000Hz** or so, which in musical terms, allows a range up to 6 octaves or so to be achieved.

- You can create collection of notes, called *playnotes* and group them into 'measures'. You can then collect your 'measures' into groups called *playmeasures*. Finally, you can collect your *playmeasures* into a final container called a *song*. You can then pass the 'song' container to a function to play the notes in the 'carefully constructed song structure'.

There is *one* single disadvantage regarding this mode, however:

- It relies on the 16-bit timer of the MCU, which is also used by the **STOPWATCH** subsystem module to support the **USONIC** module. Consequently, if you need to use the ultrasonic sensor (which means you will need the **STOPWATCH**), you CANNOT use the tone mode. **Therefore the SPKR subsystem module CANNOT be open in tone mode while the STOPWATCH is also open**. Only one can be open at time. If one is open, the other one *must* be closed. This is the most prevalent 'caveat' regarding usage of the **SPKR** subsystem module in tone mode. However, a user can cleverly write his/her program to juggle the two by deciding when to close one to open the other and back.

So in case you didn't catch the above, allow me to repeat that again:

> **NOTE:** You CANNOT use the STOPWATCH subsystem module and SPKR subsystem module (in *tone* mode) at the same time! Only one of this can be open at a time. You'll have to close *one* of the two to use the *other*!

About the *Beep* Mode

To use the **SPKR** subsystem module in *tone* mode, you have to open the **SPKR** subsystem module via `SPKR_open()` while specifying the mode to be that of *tone* as shown below:

```
SPKR_open( SPKR_BEEP_MODE )
```

The purpose of the *beep* mode is to come to the rescue when *tone* mode isn't available (because you also need to use the Ultrasonic sensor, and thus, need to start the **STOPWATCH**). The beep mode, however, is not as 'cool' as the tone mode, and so it exists merely for 'beeping's sake' – that is, to quench your thirst to emit some sort of audible cue with your CEENBoT – which is better than nothing.

When in *beep* mode:

- You can only specify *whole integer-only* frequency values: e.g., **275Hz**. Moreover, these frequencies are 'ball-park' frequencies, and so the sound you get may or may not be close to that.
- The maximum frequency is limited to **500Hz**. This limitation is due to the fact the *beep* mode runs off the existing timer service mechanism, which has a timer-tick rate of **1000Hz**, and thus the reason for the limitation, so you won't be able to generate pitches as high as you can with *tone* mode.
- Perhaps its biggest advantage is that there is no 'caveat'. Aside from needing to first open the beep mode, for use, it is *always* available and has no inter-dependencies with other module's resources, so you can 'beep' at any time to your heart's content.

Dependencies

Modular Dependencies

The **SPKR** subsystem module *must* be manually opened, regardless of the *mode*. It has no other dependencies.

Hardware Dependencies

The **SPKR** subsystem module has the following hardware dependencies:

- I/O port pin **PD7** on **PORTD** (used to toggle the *speaker* line).
- **Timer1** – The **SPKR** subsystem module relies on the 16-bit timer, which is also used by the **STOPWATCH** subsystem module (needed by the **USONIC** subsystem module), so you cannot use the **STOPWATCH** service or any module that relies on it, while using the **SPKR** subsystem module. That is, you can only have one of these open at a time – either the SPKR (in *tone* mode) or the **STOPWATCH**, bot not both.
- NO *interrupt service routines* (ISRs) related to **Timer1** may be used nor invoked once the **SPKR** service has been started, whether through the API (by way of `CBOT_ISR()`), or by bypassing it (e.g., by using the `ISR()` macro).

(*Continued on next page*)

(*Continued from previous page*)

> **Note:** The condition regarding "NO *interrupt service routines*" regarding **`Timer1`** is only applicable when you open the **`SPKR`** module in *any* mode. If you do NOT open the module, both **`Timer1`** and the corresponding ISRs are available for your *custom* use. Note that once you open the **`SPKR`** subsystem module by invoking `SPKR_open()` somewhere in your code, you've already paid the price regarding the 'highjacking' of **`Timer1`** and any corresponding ISRs, even if you eventually *close* it after you're done with it. So, you either choose to use the **`SPKR`** services in your program (which 'hogs' **`Timer1`**), or NOT at all.
>
> Consequently, *closing* the **`SPKR`** subsystem module after you're done with *will* release **`Timer1`** back to you, but it will not release the ISRs that have been 'highjacked' by the API, which means you won't be able to write your own corresponding **`Timer1`** -related ISRs.
>
> It is hoped that this limitation will be resolved in future revisions of the API.

Other Dependencies

The **`SPKR`** subsystem module internally uses functions from the mathematics library **`libm.a`**. You must now make sure that your linker includes this library – this is now discussed in the updated *"Getting Started"* guide. Please refer to this guide for instructions when setting up your *AVR Studio* project. This is a new requirement as of **`v1.02.000R`** of the API. If you forget to do this, you will get *linker* errors.

Function List Summary

- `SPKR_open()` – Used to open the **SPKR** module in *tone* mode and/or *beep* mode.
- `SPKR_close()` – Used to close the **SPKR** module's *tone* mode and/or *beep mode.*

Tone mode functions:

- `SPKR_tone()` – Function can be used to generate an *audible* tone at a specified tone frequency.
- `SPKR_play_tone()` – Function can be used to generate an *audible* tone at a specified tone frequency. You can also specify the duration of the time, and a percentage of that duration for which the tone will remain active.
- `SPKR_note()` – Function can be used to play a *musical note* at a given *octave* and *transposition*.
- `SPKR_play_note()` – Function can be used to play a *musical* note at a given *octave* and *transposition*. You can also specify the duration of time, and a percentage of that duration for which the note will remain active.
- `SPKR_play_song()` – Function can be used to play a 'song' (in *tone* mode only).
- `SPKR_map_diatone()` – Function can be used to map the numbers 0-6 to the diatonic scale.

Beep mode functions:

- `SPKR_beep()` – Function can be used to emit a *beep* at a specified frequency in *beep* mode.
- `SPKR_play_beep()` – Function can be used to emit a *beep* at a specified frequency in *beep* mode. You can also specify the duration of time, and a percentage of that duration for which the beep will remain active.

Function Reference

The `SPKR_open()` Function

Format:

```
SUBSYS_OPENSTAT SPKR_open( SPKR_MODE spkr_mode )
```

Description:

Function acquires and initializes resources needed for operation of the **SPKR** subsystem module. It can be used to open the **SPKR** subsystem module either in *tone* mode or *beep* mode. You *must* call this function first with a successful 'open' before invoking any other function provided by this module.

Input Arguments:

`spkr_mode` – Must be one of the following enumerated constants:

'`SPKR_BEEP_MODE`' – To access the *beep* mode features.
'`SPKR_TONE_MODE`' – To access the *tone* mode features.

Returns:

Returns a structure of type `SUBSYS_OPENSTAT` whose field entries indicate the status of the open request and the subsystem that resulted in an error (when, and *if* an error occurs). See *Chapter 1, Procedure for Opening Modules Before Use* for examples on how to handle the *open status*.

Example

Example shows how to open the SPKR subsystem module for both, *tone* mode and *beep* mode access:

```
#include "capi324v221.h"

void CBOT_main( void )
{

    SUBSYS_OPENSTAT ops_spkr_beep;
    SUBSYS_OPENSTAT ops_spkr_tone;

    // Attempt to open SPKR modules in both modes.
    ops_spkr_beep = SPKR_open( SPKR_BEEP_MODE );
    ops_spkr_tone = SPKR_open( SPKR_TONE_MODE );

    // Proceed only if we were able to successfully open both.
    if ( ( ops_spkr_beep.state == SUBSYS_OPEN ) &&
         ( ops_spkr_tone.state == SUBSYS_OPEN ) )
    {

        // ... DO SPEAKER STUFF ...

    } // end if()

} // end CBOT_main()
```

The `SPKR_close()` Function

Format:

```
void SPKR_close( SPKR_MODE spkr_mode )
```

Description:

Function deallocates and releases resources being used by the **SPKR** subsystem module. No other functions should be invoked once the system module is closed for the *mode* in question (e.g., *tone* mode functions, or *beep* mode functions) depending on which mode was closed.

Premature closing of the **SPKR** subsystem module in *tone* mode may sometimes be necessary to gain access to the **STOPWATCH** module which relies on the 16-bit timer, which this subsystem module also relies upon.

Input Arguments:

`spkr_mode` – Must be one of the following enumerated constants:

'`SPKR_BEEP_MODE`' – To close *beep* mode access.
'`SPKR_TONE_MODE`' – To close *tone* mode access.

Example:

Example shows a case where we're force to 'juggle' opening and closing between the **SPKR** subsystem module (tone mode) and the **STOPWATCH**.

```
// Somewhere in your program...
SUBSYS_OPENSTAT ops;

// Close the SPKR (tone mode).
SPKR_close( SPKR_TONE_MODE );

// Now open the STOPWATCH...
ops = STOPWATCH_open();

if ( ops.state == SUBSYS_OPEN )
{

    // Okay..., now open the USONIC subsystem module.
    ops = USONIC_open();

    // If good, then we can now use the USONIC.
    if ( ops.state == SUBSYS_OPEN )
    {

        // ... WE CAN NOW USE THE ULTRASONIC DEVICE ...

    } // end if()

} // end if()
```

Tone Mode Functions

The `SPKR_tone()` Function

Format:

```
void SPKR_tone( SPKR_FREQ tone_freq )
```

Description:

The `SPKR_tone()` function can be used to emit audible tone (in *tone* mode) at the specified tone frequency. The tone will remain audible until the function is invoked again with 0 frequency, which cancels the previous tone, or until `SPKR_stop_tone()` is called.

Input Arguments:

`tone_freq` – You must pass to this parameter the frequency value of the tone multiplied by **`10`**, up to a single decimal precision. To do this, you can use the `SPKR_FREQ()` helper macro-function as shown in the 'Example' section that follows below. The allowed frequency values must be between **`0`** (which cancels a previously occurring tone) and **`1000Hz`**.

Example:

The following example, assumes the SPKR subsystem module is open in *tone* mode:

```
// Emit a tone at 720.3Hz:
SPKR_tone( SPKR_FREQ( 720.3 ) );


   // .... Do some stuff....


// Sometime later...
SPKR_tone_stop();
```

Note that the first line above is equivalent to: `SPKR_tone( 7203 )`

However, using the '`SPKR_FREQ()`' macro keeps it 'natural'.

The `SPKR_play_tone()` Function

Format:

```
void SPKR_play_tone( SPKR_FREQ tone_freq, SPKR_TIME duration_ms, unsigned short int len )
```

Description:

Function works *exactly* as `SPKR_play_tone()`, except that with one function call we can specify the duration of the tone, and the percentage of *that* duration for which the tone will remain active (audible).

(*Continued on next page*)

(*Continued from previous page)*

<u>Input Arguments</u>:

`tone_freq` – You must pass to this parameter the frequency value of the tone multiplied by **10**, up to a single decimal precision. To do this, you can use the `SPKR_FREQ()` helper macro-function as shown in the 'Example' section that follows below. The allowed frequency values must be between **0** (which cancels a previously occurring tone) and **1000Hz**.

`duration_ms` – This parameter specifies the duration (in *milliseconds*) that the tone will occupy – that is, the amount of time which the function will take to complete. The duration specifies the *maximum* possible time that the tone will remain active (audible). However, the actual time for which the tone is audible could be smaller than this, as specified by the following parameter '`len`'. The value of this parameter can be anywhere from **0** to **32767ms** (~32.7 seconds).

`len` – This parameter specifies the *percentage* of the '`duration_ms`' parameter for which the note is 'audible'. The value must be anywhere between **0** and **100**.

<u>Example</u>:

The following example plays three different tones, <u>all occupying the same amount of time</u> (**250ms**). However, the first two notes are only audible for 80% of the time (**200ms**) and silent for the remaining **50ms**, and the last note is audible for 20% of the time (**50ms**) and silent for the remaining **200ms**.

```
SPKR_play_tone( SPKR_FREQ( 720.3 ), 250, 80 ); // Long tone.
SPKR_play_tone( SPKR_FREQ( 440.0 ), 250, 80 ); // Long tone.
SPKR_play_tone( SPKR_FREQ( 200.0 ), 250, 20 ); // Really short tone.
```

Note that unlike `SPKR_tone()`, you do not have to explicitly cancel the tone. The tone ends when the percentage of the duration specifies expires.

The `SPKR_note()` Function

<u>Format</u>:

```
void SPKR_note( SPKR_NOTE note, SPKR_OCTV octave, signed short int transp )
```

<u>Description</u>:

Function can be used to play a *musical note* at a specified *octave* and applied semitone *note transposition*. The note remains active until `SPKR_stop_tone()` is invoked. This is akin to playing the keys on a piano – there are 'octaves' and there are 12 tones total per octave: C, C#, D, D#, E, F, F#, G, G#, A, A#, and B. Also, like some instruments in real life, notes can be transposed to sound different than written (*let's hear it for the clarinet players – they know what I mean*).

(*Continued on next page*)

(*Continued from previous page*)

Input Arguments:

`note` – This parameter specifies the 'note' to play (e.g., the *piano key*). It must be one of the following numerical values, or equivalent enumerated constants:

0	or	SPKR_NOTE_C	(for C)
1	or	SPKR_NOTE_C_S	(for C#)
2	or	SPKR_NOTE_D	(for D)
3	or	SPKR_NOTE_D_S	(for D#)
4	or	SPKR_NOTE_E	(for E)
5	or	SPKR_NOTE_F	(for F)
6	or	SPKR_NOTE_F_S	(for F#)
7	or	SPKR_NOTE_G	(for G)
8	or	SPKR_NOTE_G_S	(for G#)
9	or	SPKR_NOTE_A	(for A)
10	or	SPKR_NOTE_A_S	(for A#)
11	or	SPKR_NOTE_B	(for B)

`octave` – Specifies the corresponding octave for the note given in 'note' above. It must be one of the following enumerated constants or equivalent numerical values:

0	or	SPKR_OCTV0	
1	or	SPKR_OCTV1	
2	or	SPKR_OCTV2	
3	or	SPKR_OCTV3	(this is considered the *middle* octave w/ Middle-C)
4	or	SPKR_OCTV4	
5	or	SPKR_OCTV5	

`transp` – This specifies the number of *semitones* for which the note will be transposed. You can transpose a note by **`+12`** semitones up, or **`-12`** semitones down.

When transposing a note, you have to make sure the resulting note it would sound like doesn't exceed beyond the maximum note. For example, if you say you want to emit the C note on octave 5 (highest octave possible) and you also transpose that by +12 semitones, well that puts you at C note on octave 6, which doesn't exist – so keep that in mind.

(*Continued on next page*)

(*Continued from previous page*)

Example:

```
// This plays middle-C.
SPKR_note( SPKR_NOTE_C, SPKR_OCTV3, 0 );

TMRSRVC_delay( TMR_SECS( 1 ) );

// This plays F#
SPKR_note( SPKR_NOTE_F_S, SPKR_OCTV3, 0 );

TMRSRVC_delay( TMR_SECS( 1 ) );

// This plays middle-C (but sounds like D, because D is two semitones
// away from C below it).
SPKR_note( SPKR_NOTE_C, SPKR_OCTV3, 2 );

TMRSRVC_delay( TMR_SECS( 1 ) );

// Stop the tone.
SPKR_stop_tone();
```

Note how in the above example we have to insert delays, to give each note some time to be audible. Sometimes you want to avoid inserting delays, however. To do this, look up `SPKR_play_note()` (next).

The `SPKR_play_note()` Function

Format:

```
void SPKR_play_note( SPKR_NOTE note, SPKR_OCTV octave, signed short int transp,
                      SPKR_TIME duration_ms,  unsigned short int len )
```

Description:

Function works exactly as `SPKR_note()`, where you can specify a *note*, *octave* and *transposition* values, but in addition, you can also specify the note's duration, and a percentage of *that* duration for which the note will remain active before it goes silent.

(*Continued on next page*)

(*Continued from previous page*)

Input Arguments:

`note` – This parameter specifies the 'note' to play (e.g., the *piano key*). It must be one of the following numerical values, or equivalent enumerated constants:

0	or	SPKR_NOTE_C	(for C)
1	or	SPKR_NOTE_C_S	(for C#)
2	or	SPKR_NOTE_D	(for D)
3	or	SPKR_NOTE_D_S	(for D#)
4	or	SPKR_NOTE_E	(for E)
5	or	SPKR_NOTE_F	(for F)
6	or	SPKR_NOTE_F_S	(for F#)
7	or	SPKR_NOTE_G	(for G)
8	or	SPKR_NOTE_G_S	(for G#)
9	or	SPKR_NOTE_A	(for A)
10	or	SPKR_NOTE_A_S	(for A#)
11	or	SPKR_NOTE_B	(for B)

`octave` – Specifies the corresponding octave for the note given in 'note' above. It must be one of the following enumerated constants or equivalent numerical values:

0	or	SPKR_OCTV0	
1	or	SPKR_OCTV1	
2	or	SPKR_OCTV2	
3	or	SPKR_OCTV3	(this is considered the *middle* octave w/ Middle-C)
4	or	SPKR_OCTV4	
5	or	SPKR_OCTV5	

`transp` – This specifies the number of *semitones* for which the note will be transposed. You can transpose a note by **+12** semitones up, or **-12** semitones down.

When transposing a note, you have to make sure the resulting note it would sound like doesn't exceed beyond the maximum note. For example, if you say you want to emit the C note on octave 5 (highest octave possible) and you also transpose that by +12 semitones, well that puts you at C note on octave 6, which doesn't exist – so keep that in mind.

`duration_ms` – This parameter specifies the duration (in *milliseconds*) that the tone will occupy – that is, the amount of time which the function will take to complete. The duration specifies the *maximum* possible time that the tone will remain active (audible). However, the actual time for which the tone is audible could be smaller than this, as specified by the following parameter '`len`'. The value of this parameter can be anywhere from **0** to **32767ms** (~32.7 seconds).

`len` – This parameter specifies the *percentage* of the '`duration_ms`' parameter for which the note is 'audible'. The value must be anywhere between **0** and **100**.

Example:

The following example plays the melodic interval C-maj (C-E-G), and then plays F-maj (F-A-C) by transposing C-maj by 5 semitones up:

```
// Play C-E-G.
SPKR_play_note( SPKR_NOTE_C, SPKR_OCTV3, 0, 250, 80 );
SPKR_play_note( SPKR_NOTE_E, SPKR_OCTV3, 0, 250, 80 );
SPKR_play_note( SPKR_NOTE_G, SPKR_OCTV3, 0, 250, 40 ); // Short staccato.

SPKR_play_note( SPKR_NOTE_NONE, SPKR_OCTV3, 0, 250, 100 ); // Silence...

// Play F-A-C by transposing C-E-G by +5 semitones up.
SPKR_play_note( SPKR_NOTE_C, SPKR_OCTV3, 5, 250, 80 );
SPKR_play_note( SPKR_NOTE_E, SPKR_OCTV3, 5, 250, 80 );
SPKR_play_note( SPKR_NOTE_G, SPKR_OCTV3, 5, 250, 40 ); // Short staccato.
```

Note that we've inserted a 'silent' note to give some spacing. We could have just as well inserted a delay here. In any case, ALL notes take 250ms to complete. However, some are audible for 80% of the time, while some are audible for 40% of the time, so they sound shorter.

The `SPKR_play_song()` Function

Format:

```
void SPKR_play_song( SPKR_SONG *pSong )
```

Description:

Function can be used to play a *song*. A *song* consists of a properly constructed structure containing various elements which are themselves other structures. To give you a general idea, you first construct a collection of notes called *playnotes*. A *playnote* consists of the following components:

<*note_value*> <*octave*> <*transposition*> <*duration_ms*> <*len*>

Note that these are the same *exact* parameters passed on to the `SPKR_play_note()` function! A *playnote* is declared via the `SPKR_PLAYNOTE` structure type. The preferred approach is to declare a collection of *playnotes* as follows:

```
SPKR_PLAYNOTE measure_1[] = {

    { SPKR_NOTE_C, SPKR_OCTV3, 0, 250, 80 },
    { SPKR_NOTE_E, SPKR_OCTV3, 0, 250, 80 },
    { SPKR_NOTE_G, SPKR_OCTV3, 0, 250, 20 }

};

SPKR_PLAYNOTE measure_2[] = {

    { SPKR_NOTE_F, SPKR_OCTV3, 0, 250, 80 },
    { SPKR_NOTE_A, SPKR_OCTV3, 0, 250, 80 },
    { SPKR_NOTE_C, SPKR_OCTV3, 0, 250, 20 }

};
```

As in *music theory* a collection of notes in a composition make up a *measure*. The above declaration declares an *array* of THREE playnotes (not just one). You can create as many of these as you want – or to be more precise, as memory will allow.

The next ingredient towards creating a *song* is to collect all your 'measures' (e.g., *group of playnotes*) together into a structure of type `SPKR_MEASURE`. The entries of this structure are laid out as follows:

<*playnote_array_name*> <*number_of_playnotes_in_array*> <*times_to_repeat*>

That is for each 'collection of *playnotes*' you have to specify the number of notes in each collection, and the number of times you want to play those notes. In the above example, we have *2* such collections, each containing 3 notes, so we collect our 'collection-of-notes' as follows:

(*Continued on next page*)

(*Continued from previous page*)

```
SPKR_MEASURE measures[] = {

    { measure_1, 3, 2 },
    { measure_2, 3, 1 }

};
```

The above declaration declares an array of *measures*. The first collection of notes has three notes in it, and we want these notes to be played twice, while the second collection also has three notes, but we want to play these notes once. In essence, and at this point, the sequence of notes played will consist of:

C-E-G-C-E-G-F-A-C

The last ingredient is the *song* structure itself, which is declared via the `SPKR_SONG` structure. It contains entries that mimic that of the `SPKR_MEASURE`:

<*playmeasure_name*> <*number_of_measures*> <*times_to_repeat*>

So the final song structure would be declared as:

```
SPKR_SONG song[] = {

    { measures, 2, 1 }

};
```

That is the *song* structure is our 'grand' collection of 'measure' containers, each containing 'containers of notes', etc. In the above example, we have one such 'grand' container, but it is possible to create many collection of notes, each one with a different musical or melodic pattern. Collect those into unique *measures* constituting different 'sections' of a musical composition, and finally collecting those 'sections' into 'movements' to make up our song.

The final step is to play the song itself, which we do by passing the '`song`' to the `SPKR_play_song()` function:

```
SPKR_play_song( song );
```

Note that *repetition* in a musical pattern can occur within two different context: 1) within each *measure* such as the one shown above when `SPKR_MEASURE` was declared; 2) within the *song* itself, such as the one shown above when `SPKR_SONG` was declared.

Lastly, the **`SPKR`** subsystem module must have been previously open via `SPKR_open()` (in *tone* mode) in order to use this function.

<u>Input Arguments</u>:

`pSong` – You must pass to this argument *THE ADDRESS OF* a structure of type `SPKR_SONG`, which must be 'carefully' constructed to contain valid song data (`SPKR_MEASURE` containing `SPKR_PLAYNOTES`, etc) as explained in the 'Description' section above.

Example:

The following example plays the first four measures of *jingle bells*:

Jin-gle-bells-pause	(measure 1)	*E-E-E-**
Jin-gle-bells-pause	(measure 2)	*E-E-E-**
Jin-gle-all-the	(measure 3)	*E-G-C-D*
Waaaaay	(measure 4)	*E---------*

Such a song could be constructed as shown by the following comprehensive listing:

```
// Auth: Jose Santos
// Desc: Sample program shows how to construct a 'song' by playing Jingle Bells.

#include "capi324v221.h"

// ------------------------------------ Global Data:

// Meausure 1 & 2 (we can repeat this)
SPKR_PLAYNOTE measure_1_2[] = {

    { SPKR_NOTE_E, SPKR_OCTV3, 0, 250, 80 },
    { SPKR_NOTE_E, SPKR_OCTV3, 0, 250, 80 },
    { SPKR_NOTE_E, SPKR_OCTV3, 0, 250, 80 },
    { SPKR_NOTE_E, SPKR_OCTV3, 0, 250, 0  }

};

// Measure 3
SPKR_PLAYNOTE measure_3[] = {

    { SPKR_NOTE_E, SPKR_OCTV3, 0, 250, 80 },
    { SPKR_NOTE_G, SPKR_OCTV3, 0, 250, 80 },
    { SPKR_NOTE_C, SPKR_OCTV3, 0, 250, 80 },
    { SPKR_NOTE_D, SPKR_OCTV3, 0, 250, 80 }

};

// Mesure 4
SPKR_PLAYNOTE measure_4[] = {

    { SPKR_NOTE_E, SPKR_OCTV3, 0, 750, 100 }

};
```

(*Continued on next page*)

(*Continued from previous page*)

```
// Collect our measures together.
SPKR_MEASURE measures[] = {

    { measure_1_2, 4, 2 },   // Play measure_1_2 twice.
    { measure_3,   4, 1 },   // Play once.
    { measure_4,   1, 1 }    // Play once.

};

// Collect our collection of measures into a song.
SPKR_SONG song[] = {

    { measures, 3, 1 }  // There are 3 playmeasures inside of 'measures'; play once.

};

// ------------------------------------- Main:

void CBOT_main( void )
{

    // Open needed modules -- we'll just assume they open successfully.
    LCD_open();
    SPKR_open( SPKR_TONE_MODE );

    LCD_printf( "Playing Song...\n" );

    // Play the song.
    SPKR_play_song( song );

    LCD_printf( "Done.\n" );

    // Never leave.
    while( 1 );

} // end CBOT_main()
```

...and *there you have it*!

The `SPKR_map_diatone()` Function

Format:

```
SPKR_NOTE SPKR_map_diatone( SPKR_DNOTE dia_note )
```

Description:

This function maps the numbers 0 to 7 to the *diatonic* notes of the *chromatic scale*. Think of this as only dealing with the 'white keys' on a piano, while ignoring the 'black ones'. Remember that a 'SPKR_NOTE' takes a value from 0 to 11 corresponding to all 12 tones of the chromatic scale. So, if you wanted to deal with only the diatonic scale using the numerical values of the chromatic scale you would have to deal specifically with notes: 0, 2, 4, 5, 7, 9, 11. This numerical sequence, however is not linear. Thus, the function above performs the following mapping from the *diatonic* numeric representation to the *chromatic* numeric representation:

(DO) **0** maps to note **0** on the chromatic scale.
(RE) **1** maps to note **2** on the chromatic scale.
(MI) **2** maps to note **4** on the chromatic scale.
(FA) **3** maps to note **5** on the chromatic scale.
(SO) **4** maps to note **7** on the chromatic scale.
(LA) **5** maps to note **9** on the chromatic scale.
(TI) **6** maps to note **11** on the chromatic scale.

Input Arguments:

`dia_note` – A note corresponding to the notes of a diatonic scale. It can be one of the following enumerated constants, or equivalent numerical values:

`0` or `SPKR_DNOTE_C`
`1` or `SPKR_DNOTE_D`
`2` or `SPKR_DNOTE_E`
`3` or `SPKR_DNOTE_F`
`4` or `SPKR_DNOTE_G`
`5` or `SPKR_DNOTE_A`
`6` or `SPKR_DNOTE_B`

Note that '`dia_note`' is of type `SPKR_DNOTE`, while the *chromatic* note is of type `SPKR_NOTE` – there is *difference*!

Returns:

Function returns the numerical equivalent value as applicable on the *chromatic* scale (note of type `SPKR_NOTE`).

(*Continued on next page*)

(*Continued from previous page*)

Example:

The following example shows one instance on how you would apply the above function. We want to be able to play C-E-G-F-G-A-B-C sequentially ending on octave 3 and finishing on C on octave 4 with no transposition. Moreover, we'd like to do this easily in loop, which we do as follows:

```
unsigned short int i;

for( i = 0; i < 8; ++i )
{

    if ( i != 7 )

        SPKR_play_note( SPKR_map_diatone( i ), SPKR_OCTV3, 0, 250, 80 );

    else

        SPKR_play_note( SPKR_map_diatone( i ), SPKR_OCTV4, 0, 250, 80 );

} // end for()
```

It is for things like this that the function exists.

Beep Mode Functions

The `SPKR_beep()` Function

Format:

```
void SPKR_beep( SPKR_FREQ beep_frequency )
```

Description:

Function is used to generate a 'beep' of arbitrary 'ball-park' frequency. It requires that the **SPKR** subsystem module be opened in *beep* mode prior to use. The 'beep' remains active until the function is invoked again with `0` frequency value, or until `SPKR_stop_beep()` is invoked.

Input Arguments:

`beep_frequency` – This specifies the 'ball-park' frequency to emit the 'beep' at. It must be an *integer*-only value between **0** (which cancels a previously issued 'beep') and **500Hz**. It is possible to supply values larger than **500Hz** for some interesting results, but you won't actually get a 'sound' whose frequency is greater than **500Hz**.

Also, unlike the *tone* mode functions, which require frequency values to be scaled by **10** (via `SPKR_FREQ()` macro-function, you DO NOT need to scale your frequency values when invoking this function, and thus, you DO NOT need to invoke the `SPKR_FREQ()` macro as shown in some other examples.

Example:

Simple beep pattern:

```
unsigned short int i = 0;

for( i = 0; i < 3; ++i )
{

    // Beep ON.
    SPKR_beep( 440 );

    // Wait 100ms.
    TMRSRVC_delay( 100 );

    // Beep OFF.
    SPKR_beep( 0 );

    // Wait before beeping again...
    TMRSRVC_delay( 10 );

} // end for()
```

The example, of course, assumes you have already opened the **SPKR** subsystem module in *tone* mode via `SPKR_open()`.

The `SPKR_play_beep()` Function

Format:

```
void SPKR_play_beep( SPKR_FREQ beep_freq, SPKR_TIME duration_ms, unsigned short int len )
```

Description:

Function works *exactly* as `SPKR_beep()`, except that with one function call we can specify the duration of the beep, and the percentage of *that* duration for which the beep will remain active (audible).

Input Arguments:

`beep_frequency` – This specifies the 'ball-park' frequency to emit the 'beep' at. It must be an *integer*-only value between **0** (which cancels a previously issued 'beep') and **500Hz**. It is possible to supply values larger than **500Hz** for some interesting results, but you won't actually get a 'sound' whose frequency is greater than **500Hz**.

Also, unlike the *tone* mode functions, which require frequency values to be scaled by **10** (via `SPKR_FREQ()` macro-function, you DO NOT need to scale your frequency values when invoking this function, and thus, you DO NOT need to invoke the `SPKR_FREQ()` macro as shown in some other examples.

`duration_ms` – This parameter specifies the duration (in *milliseconds*) that the beep will occupy – that is, the amount of time which the function will take to complete. The duration specifies the *maximum* possible time that the beep will remain active (audible). However, the actual time for which the beep is audible could be smaller than this, as specified by the following parameter '`len`'. The value of this parameter can be anywhere from **0** to **32767ms** (~32.7 seconds).

`len` – This parameter specifies the *percentage* of the '`duration_ms`' parameter for which the beep is 'audible'. The value must be anywhere between **0** and **100**.

Chapter 10: The STEP (Stepper) Subsystem Module

> This chapter introduces you to the functional services made available by the **`STEP`** subsystem module, which provides services for controlling the CEENBoT's *stepper* motors and give the CEENBoT mobility.

Module at a Glance

Description

The **STEP** subsystem module allows users to control the CEENBoT's *stepper* motors, which give mobility to the CEENBoT. The CEENBoT currently has two stepper motors with rotational resolution of 1.8-degrees per step for a total 200 steps for a 360-degree rotational coverage. The **STEP** module is the the most complex element in the API as it provides a substantial set of features. Stepper motors can be operated in a variety of operating modes called *run modes*, and the parameters of each stepper motor such as *speed*, *linear acceleration*, *direction*, *braking modes* among other features can be controlled via this subsystem module.

Modular Dependencies

The **STEP** module *must* be manually opened by the user. It has no other modular dependencies.

Hardware Dependencies

- I/O Port pin **PC2** on **PORTC** (for 1A pin)
- I/O Port pin **PC3** on **PORTC** (for 1B pin)
- I/O Port pin **PC4** on **PORTC** (for 1C pin)
- I/O Port pin **PC5** on **PORTC** (for 2A pin)
- I/O Port pin **PC6** on **PORTC** (for 2B pin)
- I/O Port pin **PC7** on **PORTC** (for 2C pin)

In addition, the following MCU-specific dependencies exist:

- **Timer0** – The STEP module is entirely driven (*clocked*) by this timer. **Timer0** is reserved for the API and is NOT available to the user when programming CEENBoT-API code even if the *stepper* module is not loaded.

- NO *interrupt service routines* (ISRs) related to **Timer0** may be used nor invoked for the same reasons, whether through the API (by way of CBOT_ISR()), or outside of it by bypassing it (e.g., by using the ISR() macro).

Function List Summary

The Function list in the **STEPPER** module can be presently grouped into the following four categories:

- Open/close functions (you gotta' have these, obviously!).
- Functions that can be used to *set* or *get* **STEPPER** module parameters.
- Functions that either *initiate* or *cancel* a stepper 'motion'.
- Functions that can be used to *wait* on a stepper 'motion' to complete and perhaps do something about it once this happens (under specific *run modes*).

Open/Close Functions:

- **STEPPER_open()** - Opens and initializes the **STEPPER** subsystem module.
- **STEPPER_close()** - Closes and relinquishes the **STEPPER** subsystem module.

Get/Set Functions:

- `STEPPER_set_mode()` - Sets the the operating *run mode* (*free-running* or *step* mode) for the steppers.
- `STEPPER_set_pwr_mode()` - Sets the operating *power mode*.
- `STEPPER_set_speed()` - Sets the stepper's *speed*. Although categorized as a 'set' function, it also happens to be a *motion-initiating* function, so it's discussed in *that* category.
- `STEPPER_set_accel()` - Sets the stepper's *linear acceleration*.
- `STEPPER_set_dir()` - Sets the stepper's rotational *direction*.
- `STEPPER_set_stop_mode()` - Sets the *stop mode* upon completion of a stepping sequence when the operating *run mode* is *step* mode.
- `STEPPER_set_steps()` - Sets the number of *steps* a stepper should move when the operating *run mode* is *step* mode.
- `STEPPER_get_curr_speed()` - Get the current speed of the stepper motors.
- `STEPPER_get_nSteps()` - Get the number of remaining steps when the operating *run mode* is *step mode*.

Motion Initiating/Canceling Functions:

- `STEPPER_set_speed()` - Sets the stepper's *speed*.
- `STEPPER_run()` - Issue stepper motion when the operating *run mode* is *free-running*.
- `STEPPER_step()` - Issue a stepper motion when the operating *run mode* is *step*.
- `STEPPER_stepwt()` - Issue a stepper motion when *run mode* is *step mode* (BLOCKING version).
- `STEPPER_stepnb()` - Issue a stepper motion when *run mode* is *step mode* (NON-BLOCKING version).
- `STEPPER_move()` - Issue a stepper motion with independent controls for left and right motor in one of the three possible operating run modes *free-running, step-block* and *step-no-block*.
- `STEPPER_move_rn()` - Issue a '`STEPER_move()`' with *run mode* set to *free-running*.
- `STEPPER_move_stnb()` - Issue a '`STEPPER_move()`' with *run mode* set to *step-no-block*.
- `STEPPER_move_stwt()` - Issue a '`STEPPER_move()`' with *run mode* set to *step-block*.
- `STEPPER_stop()` - Stop the stepper motors and possibly engage the brakes.
- `STEPPER_go()` - Specifically used to *remove* brakes and continue motion after a previously issued '`STEPPER_stop()`' with the brakes ON.

'Wait' Functions:

- `STEPPER_wait_on()` - When the operating *run mode* is one of the *step* modes this can be used to 'wait' for one (or both) of the steppers to complete its motion before doing anything else.
- `STEPPER_wait_and_then()` - When the operating *run mode* is one of the *step* modes this can be used to 'wait' for one (or both) of the steppers to complete its motion before doing anything else, and in addition, trigger a special function called a *stepper event* once the motion completes.

Function Reference: Open/Close Functions

The `STEPPER_open()` Function

Format:

```
SUBSYS_OPENSTAT STEPPER_open( void )
```

Description:

Function acquires and initializes resources needed for operation of the **STEPPER** subsystem module. You *must* call this function first with a successful 'open' before invoking any other function provided by this module.

Returns:

Returns a structure of type `SUBSYS_OPENSTAT` whose field entries indicate the status of the open request and the subsystem that resulted in an error (when, and *if* an error occurs). See *Chapter 1, Procedure for Opening Modules Before Use* for examples on how to handle the *open status*.

Example:

```
#include "capi324v221.h"

void CBOT_main( void )
{

  SUBSYS_OPENSTAT opstat;

  // Open the STEPPER module.
  opstat = STEP_open();

  if ( opstat.state == SUBSYS_OPEN )
  {

        // ... DO STEPPER STUFF ...

  }// end if()

} // end CBOT_main()
```

The `STEPPER_close()` Function

Format:

```
void STEPPER_close( void )
```

Description:

Function deallocates and releases resources being used by the **STEPPER** subsystem module. No other functions should be invoked once the subsystem module is closed.

Function Reference: Get/Set Functions

The *get/set* category of functions allow the setting (or retrieval) of parameters that control the behavior of the stepper engine when motion is initiated via the motion-initiating functions.

The `STEPPER_set_mode()` Function

Format:

```
void STEPPER_set_mode( STEPPER_ID which, STEPPER_MODE mode )
```

Description:

This function allows you to set the operating *run mode* for each of the stepper motors. See the section on *motion-initiating/canceling* functions for an explanation of operational *run modes*.

Input Arguments:

`which` – Must be one of the following enumerated constants that specifies *which* stepper is to be affected by the 'set' operation:

`STEPPER_LEFT` (or `LEFT_STEPPER`) – Only the *left* stepper will be affected.
`STEPPER_RIGHT` (or `RIGHT_STEPPER`) – Only the *right* stepper will be affected.
`STEPPER_BOTH` (or `BOTH_STEPPERS`) – Both steppers will be affected.

`mode` – Must be one of the following enumerated constants:

`STEPPER_FREERUNNING_MODE` (or `STEPPER_NORMAL_MODE`) - Set the specified stepper to run in *free-running* mode.

`STEPPER_STEP_MODE` – Set the specified stepper to run in *step* mode. See the section on *motion-initiating/canceling* functions for an explanation on operating *run modes*.

Example:

Refer to the example given for `STEPPER_set_stop_mode()`.

The `STEPPER_set_pwr_mode()` Function

Format:

```
void STEPPER_set_pwr_mode( STEPPER_PWRMODE power_mode )
```

Description:

This function can be used to set the *power mode* for BOTH stepper motors – that is, *all steppers* will be affected by this operation. Stepper motors by their nature can use a lot of power, draw a lot of current and drain your battery quickly. Therefore, by default the steppers are operated in *low power* mode which is sufficient for most uses and the energy savings are quite significant. However, the downside to operating the steppers in low power mode is reduced *torque*. If you find the need to use the steppers at their full potential, then you can use this function to engage the *high power* mode. However, you are cautioned to use this mode judiciously as your batteries will drain much quickly in high power mode.

(*Continued on next page*)

(*Continued from previous page*)

Input Arguments:

`power_mode` – Specify for this argument one of the following enumerated constants:

> `STEPPER_PWR_LOW` – Set the steppers to run in *low power* mode (default).
> `STEPPER_PWR_HIGH` – Set the steppers to run in *high power* mode. (Best get your *charger* ready).

The `STEPPER_set_speed()` Function

> **Note:** Because this function also happens to be a *motion-initiating* function it is discussed in *that* category. See the section on *Motion Initiating/Canceling Functions* in this chapter for information on this function.

The `STEPPER_set_accel()` Function

Format:

```
void STEPPER_set_accel( STEPPER_ID which, unsigned short int accel_rate )
```

Description:

This function allows you to set the *linear acceleration* (and consequently also the *deceleration*) for the specified stepper motor(s). When the specified acceleration is non-zero, you're effectively enabling the *linear acceleration* feature. This means that when you initiate a motion to a specified speed, the steppers will gradually accelerate to that speed at the specified acceleration rate. However, if you specify zero acceleration, you're essentially disabling the linear acceleration feature. There are circumstances where you might want to enable linear acceleration or disable it – for example, if you're implementing a PID loop and you are manually controlling the speed of the stepper motors – in that case *you* are in control of acceleration and so it would be best to disable linear acceleration.

When linear acceleration is enabled, it also applies to deceleration. That is, when the motors speed is suddenly set to zero, the speed will decrease at the same rate as that specified for acceleration. Presently there is no support for specifying acceleration independent of deceleration, but this might change in the future.

Acceleration is specified in *steps/sec*^{2}.

Input Arguments:

`which` – Must be one of the following enumerated constants that specifiees *which* stepper is to be affected by the 'set' operation:

> `STEPPER_LEFT` (or `LEFT_STEPPER`) – Only the *left* stepper will be affected.
> `STEPPER_RIGHT` (or `RIGHT_STEPPER`) – Only the *right* stepper will be affected.
> `STEPPER_BOTH` (or `BOTH_STEPPERS`) – Both steppers will be affected.

`accel_rate` – Specify the acceleration rate (in *steps/sec*^{2}) for the specified motor(s). The acceleration must be in the range of `0` to `1000`, with `0` meaning that *linear acceleration/deceleration* is effectively turned OFF.

(*Continued on next page*)

(*Continued from previous page*)

Example:

Refer to the example given for `STEPPER_set_speed()` in the motion-initiating/canceling functions section.

The `STEPPER_set_dir()` Function

Format:

```
void STEPPER_set_dir( STEPPER_ID which, STEPPER_DIR dir )
```

Description:

This function sets the direction the stepper motors will move once in motion (specifically, when you specify a *non-zero* speed via `STEPPER_set_speed()`).

Input Arguments:

`which` – Must be one of the following enumerated constants that specifies *which* stepper is to be affected by the 'set' operation:

- `STEPPER_LEFT` (or `LEFT_STEPPER`) – Only the *left* stepper will be affected.
- `STEPPER_RIGHT` (or `RIGHT_STEPPER`) – Only the *right* stepper will be affected.
- `STEPPER_BOTH` (or `BOTH_STEPPERS`) – Both steppers will be affected.

`dir` – Specifies the direction for the corresponding stepper(s). It must be one of the following enumerated constants:

- `STEPPER_FWD` – Stepper(s) will move *forward*.
- `STEPPER_REV` – Stepper(s) will move in *reverse*.

Example:

See the example given for `STEPPER_set_stop_mode()`.

The `STEPPER_set_stop_mode()` Function

Format:

```
void STEPPER_set_stop_mode( STEPPER_ID which, STEPPER_BRKMODE brakeMode )
```

Description:

When the steppers *run mode* is set to *step* mode, this function can be used to set whether braking will take place once the stepping motion completes for the stepper motor(s) in question. By default when the steppers are commanded to move a finite number of steps and the motion completes, the stepper motors become idle. However, with this function you can set the mode so that when a stepper motion completes, the brakes will be immediately enabled.

(*Continued on next page*)

(*Continued from previous page*)

> **Note:** Note that once brakes are enabled they will remain so until the user manually disengages the brakes by issuing a call to `STEPPER_stop()`., which is a completely different function. The `STEPPER_set_stop_mode()` merely sets the *stop mode*, it doesn't engage nor disengages the brakes in any way. For engaging or disengaging the brakes, see `STEPPER_stop()`. Also see `STEPPER_go()`, which is related.

> **Note:** The `STEPPER_set_stop_mode()` function is only applicable when the operating *run mode* of the stepper(s) in question is *step* mode and NOT *free-running* mode. Otherwise, this setting is ignored.

Input Arguments:

`which` – Must be one of the following enumerated constants that specifies *which* stepper is to be affected by the 'set' operation:

`STEPPER_LEFT` (or `LEFT_STEPPER`) – Only the *left* stepper will be affected.
`STEPPER_RIGHT` (or `RIGHT_STEPPER`) – Only the *right* stepper will be affected.
`STEPPER_BOTH` (or `BOTH_STEPPERS`) – Both steppers will be affected.

`brakeMode` – You must specify one of the following enumerated constants:

`STEPPER_BRK_OFF` – Steppers will remain *idle* once a motion completes (while in *step* mode).
`STEPPER_BRK_ON` – Steppers will engage *brake* once a motion completes (while in *step* mode).

Example:

The following example sets the *run mode* for both steppers to *step* mode. Then sets the direction, number of steps the steppers should move and the stop mode. After that, the motors are set in motion by simply setting a *non-zero* speed (see `STEPPER_set_speed()` in this chapter for details). Then the timer service module is used to delay for 5 seconds (see the **TMRSRVC** module chapter for details) to allow the motor to proceed through the number of steps specified and after that, the brakes are turned off via a call to `STEPPER_stop()`.

```
// Assume that the STEPPER module has been properly opened.

// Set the operating mode to 'step' mode.
STEPPER_set_mode( STEPPER_BOTH, STEPPER_STEP_MODE );

// Set the direction the motors will move.
STEPPER_set_dir( STEPPER_BOTH, STEPPER_FWD );

// Set the number of steps that the steppers will move.
STEPPER_set_steps( STEPPER_BOTH, 200 );  // Note: 200 = 1 revolution.

// Set the stop mode.  We want the brakes to come on once the motion
// completes (i.e., when all 200 steps are advanced).
STEPPER_set_stop_mode( STEPPER_BOTH, STEPPER_BRK_ON );
```

(*Continued on next page*)

(*Continued from previous page*)

```
// Initiate the motion by setting a non-zero speed.
STEPPER_set_speed( STEPPER_BOTH, 200 );      // Note: 200 steps/sec.

// Wait five seconds for the motion to complete - 5 seconds should be enough.
TMRSRVC_delay( TMR_SECS( 5 ) );

// At this point the brakes should still be ON.  We don't want
// to drain our battery so let's turn the brakes OFF.
// NOTE:  This is a DIFFERENT function.
STEPPER_stop( STEPPER_BOTH, STEPPER_BRK_OFF );
```

The `STEPPER_set_steps()` Function

Format:

`void STEPPER_set_steps( STEPPER_ID which, unsigned short int nSteps )`

Description:

Use this function to set the number of finite *steps* the specified stepper(s) is expected to travel once the *speed* is set to a non-zero value. This setting is only applicable when the operating *run mode* is *step* mode. Otherwise it has no effect (in *free-running* mode).

Input Arguments:

`which` – Must be one of the following enumerated constants that specifies *which* stepper is to be affected by the 'set' operation:

`STEPPER_LEFT`	(or `LEFT_STEPPER`)	– Only the *left* stepper will be affected.
`STEPPER_RIGHT`	(or `RIGHT_STEPPER`)	– Only the *right* stepper will be affected.
`STEPPER_BOTH`	(or `BOTH_STEPPERS`)	– Both steppers will be affected.

`nSteps` – This is the number of steps. The maximum value is **`65535`** – as of platform '324 v2.21 of the CEENBoT, that provides about ~327 *revolutions*.

Example:

See the example given for `STEPPER_set_stop_mode()`.

The `STEPPER_get_curr_speed()` Function

Format:

`STEPPER_SPEED STEPPER_get_curr_speed( void )`

Description:

This function can be used to query the stepper engine what the *current speed* of both steppers are. Note that this is not the speed that you set via `STEPPER_set_speed()`, but the speed at which the steppers are *moving now*, regardless of the value you set via `STEPPER_set_speed()`. Capiche?

(*Continued from previous page*)

Returns:

This function returns a structure of type `STEPPER_SPEED`, which has the following form:

```
typedef struct STEPPER_SPEED_TYPE {

    signed short int left;        // Holds left stepper 'step speed'.
    signed short int right;       // Holds right stepper 'step speed'.

} STEPPER_SPEED;
```

As you can see, the structure contains two fields, so you get the speed for *both* motors.

Example:

The following code snippet shows you how you wait for the motors to reach zero speed before doing anything else:

```
// Assume the STEPPER module is already properly opened.

// Assume the motors are already moving (with non-zero speed)!

STEPPER_SPEED curr_speed;

// Wait for BOTH motors to reach zero speed -- just because you CAN!
do {

  curr_speed = STEPPER_get_curr_speed();

} while( ( curr_speed.left !=0 ) || ( curr_speed.right != 0 ) );

// Okay!  Now we can continue...

foo();
bar();
etc();
```

The `STEPPER_get_nSteps()` Function

Format:

`STEPPER_STEPS STEPPER_get_nSteps( void )`

Description:

Function can be used to query what is the number of *steps* that each stepper still has to complete. Note that this is not the value that you may have set previously with `STEPPER_set_steps()`, but what is *left* or *remaining* while motion is in progress – assuming the steppers are moving at all.

(*Continued on next page*)

(*Continued from previous page*)

Returns:

Function returns a structure of type `STEPPER_STEPS` containing two fields. It has the following form:

```
typedef struct STEPPER_STEPS_TYPE {

    unsigned short int left;  // Holds number of steps for left motor.
    unsigned short int right; // Holds number of steps for right motor.

} STEPPER_STEPS;
```

As you can see, the structure contains two fields, where the number of steps remaining to advance is given for each motor.

Example:

Very similar to the example given for `STEPPER_get_curr_speed()` – see the example for that.

Function Reference: Motion-Initiating/Canceling Functions

We now discuss functions that can initiate or cancel a give motion. However, before we go over the function details, we need to cover some fundamental concepts that you need to know to help you understand how the **STEPPER** engine does things. It is important that you read this section.

The Operating *Run Modes*

The **STEPPER** module can operate the stepper motors in two principal *run modes.* The run modes are:

- *free-running* mode
- *step* mode

You can set the run mode independently for each stepper by invoking `STEPPER_set_mode()` function previously discussed so long you don't use a higher-level stepper function that doesn't override it (the *run mode*). Furthermore, when in *step* mode, some functions make a distinction between the following execution behavior:

- *step-block* mode
- *step-no-block* mode

The free-running mode is used when you know when you want to start the motion, but have no idea when it will stop – perhaps some other condition will stop them, like IR Sensors. Consequently, when you *initiate* a motion in free-running mode, the steppers will happily keep running until you explicitly stop them! You specify parameters such as *speed* and *direction*, but there is no mention of *distance*.

On the other hand, the *step* mode is used for *finite distance moves*. That is, you know *precisely* how far you want the motor(s) to move, with this 'distance' being specified as a finite number of *steps* – hence the name '*step'* mode. This is, of course, in addition to *speed*, *direction*, *acceleration* and what have you.

Now, as already pointed out, when in *step* mode – some functions and macro-functions can operate in one of the following being *step-block* mode and *step-no-block* mode.

Functions that operate in *step-block* mode will BLOCK – that is, execution will *hold* – at the corresponding blocking function until the stepper motion completes – in other words, until the number of steps remaining to advance is exhausted to zero. Only then, will execution move on to the next following function, if any.

In contrast, functions that operate in *step-no-block* mode will NOT BLOCK – that is, execution will happen pretty much instantaneously – at the corresponding function call – and execution will continue down your program whether the motion is still way in effect or not.

Obviously, the various execution behavior modes for the step modes exist to give you flexibility. For example, the *step-blocking* functions could be used to construct complex motions by invoking a motion one after the other, in a predetermined, but controlled sequence. You may, for instance, write your own function called `avoid_obstacle()`, which calls a series of primitive *step-blocking* functions to ensure each one is executed sequentially through completion. This scenario would be perfect for issuing *step-blocking* functions. On the other hand, you may have a need to do something else immediately after you issue a motion – perhaps in an infinite loop so your functions should not block – otherwise nothing else happens. In this scenario, the *step-no-block* functions would be a best fit.

So, the point is – you *have* options – *several* of them in fact. How you approach a problem is entirely up to you and your imagination, so there are creative ways to solve a particular motion problem using a variety of tactical methods.

Understanding *Stepper Events*

The last concept to discuss regards what are called *stepper events*. A *stepper event* is just a function that can be invoked or triggered once a motion completes. This can sometimes be advantageous in contrast to 'busy-waiting' for a stepper motion to compete. For this reason, stepper events are only useful when you initiate a motion in the *step-no-block* modes.

You declare (provide a prototype) and define (provide function body) a stepper event just like any other function, but you must do so with the `STEPPER_EVENT()` macro. For example:

```
// ------------ prototype:
STEPPER_EVENT( when_left_finishes  );
STEPPER_EVENT( when_right_finishes );

// ------------ functions:
STEPPER_EVENT( when_left_finishes ) {

  LCD_printf( “Left stopped!\n” );

}

STEPPER_EVENT( when_right_finishes ) {

  LCD_printf( “Right stopped!\n” );

}
```

You can then provide the the name of your stepper event functions to either `STEPPER_move()`, or `STEPPER_wait_and_then()` functions. See the documentation and examples sections for these respective functions to see how you might apply stepper events.

Now that you're 'in-the-know' with the *run modes* and what *stepper events* are – we can begin covering the functions in detail. The functions are listed from low-level to higher-complexity functions – *check it!*

The `STEPPER_set_speed()` Function

Format:

```
void STEPPER_set_speed( STEPPER_ID which, unsigned short int nStepsPerSec )
```

Description:

This is the most *primitive* motion-initiating function – in fact many functions that follow internally invoke this function at some point. Think of this function as the 'gas-pedal' for the stepper motors – except instead of just having one 'gas pedal' you have two! – one for each of the steppers. Essentially, as soon as the speed is *non-zero*, the motors are going to move. If the operating *run mode* is *free-running*, it just goes. If it's *step* mode, then it will move ONLY if it has a finite number of steps remaining to advance, which you must set via `STEPPER_set_steps()` function. The *direction* and *acceleration* (if enabled) should be set with the appropriate function calls.

> **Note:** Make sure your brakes are not engaged when you issue a move in *free-running* mode. Would your car move while you have your feet on both, the brake and gas pedals? Well – neither would the CEENBoT. See `STEPPER_stop()` function to learn how to disengage (and engage) the brake. Also see the related function `STEPPER_go()`.

Input Arguments:

`which` – Must be one of the following enumerated constants that specifies *which* stepper is to be affected by the 'set' operation:

`STEPPER_LEFT`	(or `LEFT_STEPPER`)	– Only the *left* stepper will be affected.
`STEPPER_RIGHT`	(or `RIGHT_STEPPER`)	– Only the *right* stepper will be affected.
`STEPPER_BOTH`	(or `BOTH_STEPPERS`)	– Both steppers will be affected.

`nStepsPerSec` – This is the speed in *steps/sec*. The maximum rate is capped at **~400** *steps/sec* – that's approximately 2 *revolutions/sec*. Anything higher makes the *torque* become non-existent and the steppers begin to rattle erratically.

Example:

Here's a quick example – showcasing several of the *set* functions also:

```
// Assume the STEPPER module has been properly opened.

// Set the operating 'run-mode'.
STEPPER_set_mode( STEPPER_BOTH, STEPPER_FREERUNNING_MODE );

// Set the direction.
STEPPER_set_dir( STEPPER_BOTH, STEPPER_FWD );

// Set the acceleration to 400 steps/sec^2 -- just for fun.
STEPPER_set_accel( STEPPER_BOTH, 400 );

// GO!...  forever @ 150 steps/sec.
STEPPER_set_speed( STEPPER_BOTH, 150 );
```

The `STEPPER_run()` Function

Format:.

`void STEPPER_run( STEPPER_ID which, STEPPER_DIR dir, unsigned short int nStepsPerSec )`

Description:

This is a slightly less *primitive*, compound, motion-initiating function. It essentially combines a call to `STEPPER_set_mode()` to be set to *free-running*, `STEPPER_set_dir()` to set the *direction*, and `STEPPER_set_speed()` to set the speed and allow motion to take place – all with one single function call. It is a *convenience* function.

Input Arguments:

`which` – Must be one of the following enumerated constants that specifies *which* stepper is to be affected by the 'set' operation:

- `STEPPER_LEFT` (or `LEFT_STEPPER`) – Only the *left* stepper will be affected.
- `STEPPER_RIGHT` (or `RIGHT_STEPPER`) – Only the *right* stepper will be affected.
- `STEPPER_BOTH` (or `BOTH_STEPPERS`) – Both steppers will be affected.

`dir` – Specifies the direction for the specified stepper(s). It must be one of the following enumerated constants:

- `STEPPER_FWD` – Stepper(s) will move *forward*.
- `STEPPER_REV` – Stepper(s) will move in *reverse*.

`nStepsPerSec` – This is the speed in *steps/sec*. The maximum rate is capped at **~400** *steps/sec* – that's approximately 2 *revolutions/sec*. Anything higher makes the *torque* become non-existent and the steppers begin to rattle erratically.

Example:

The example shown for `STEPPER_set_speed()` can be simplified much better using this function instead:

```
// Assume the STEPPER module has been properly opened.

// Set the acceleration to 400 steps/sec^2 -- just for fun.
STEPPER_set_accel( STEPPER_BOTH, 400 );

// GO! -- Both Steppers, Forward, @ 150 steps/sec in 'free-running' mode.
STEPPER_run( STEPPER_BOTH, STEPPER_FWD, 150 );
```

MUCH simpler, and why it is called a *convenience* function.

The `STEPPER_step()` Function

Format:

```
void STEPPER_step(    STEPPER_ID            which,
                      STEPPER_DIR           dir,
                      unsigned short int    nSteps,
                      unsigned short int    nStepsPerSec,
                      STEPPER_BRKMODE       onStopDowhat,
                      STEPPER_WAITMODE      wait_mode,
                      STEPPER_NOTIFY        *pNotifyFlag )
```

Description:

This function allows you to initiate a stepper motion in *step* mode. It is the analog to `STEPPER_run()`, but specific to the *step* mode – so it has more parameters you can specify. It is a convenience function which internally sets the operating mode to *step* mode (by calling `STEPPER_set_mode()`), the *direction* (via `STEPPER_set_dir()`), distance in *steps* (via `STEPPER_set_steps()`), speed (via `STEPPER_set_speed()`), *brake mode* (via `STEPPER_set_stop_mode()`), the *wait* mode and specify a *notification flag*. It accomplishes this by calling these aforementioned primitive functions, so it is a *compound* function.

> **Note:** You are free to use this function. However, I highly recommend that you take a look at `STEPPER_setwt()` and `STEPPER_stepnb()` *macro-functions*. They're much better as they avoid some parameters that are usually not necessary.

Input Arguments:

`which` – Must be one of the following enumerated constants that specifies *which* stepper is to be affected by the 'set' operation:

`STEPPER_LEFT` (or `LEFT_STEPPER`) – Only the *left* stepper will be affected.
`STEPPER_RIGHT` (or `RIGHT_STEPPER`) – Only the *right* stepper will be affected.
`STEPPER_BOTH` (or `BOTH_STEPPERS`) – Both steppers will be affected.

`dir` – Specifies the direction for the specified stepper(s). It must be one of the following enumerated constants:

`STEPPER_FWD` – Stepper(s) will move *forward*.
`STEPPER_REV` – Stepper(s) will move in *reverse*.

`nSteps` – This is the number of steps. The maximum value is **`65535`** – as of platform '324 v2.21 of the CEENBoT, that provides about ~327 *revolutions*.

`nStepsPerSec` – This is the speed in *steps/sec*. The maximum rate is capped at **`~400`** *steps/sec* – that's approximately 2 *revolutions/sec*. Anything higher makes the *torque* become non-existent and the steppers begin to rattle erratically.

`onStopDowhat` – You must specify one of the following enumerated constants. See the info on `STEPPER_set_stop_mode()` function for the meaning of this parameter:

`STEPPER_BRK_OFF` – Steppers will remain *idle* once a motion completes (while in *step* mode).
`STEPPER_BRK_ON` – Steppers will engage *brake* once a motion completes (while in *step* mode).

(*Continued on next page*)

(*Continued from previous page*)

`wait_mode` – This determines whether the function's execution model will behave as a *step-block* (BLOCKING) function or as a *step-no-block* (NON-BLOCKING) function. You must specify one of the following enumerated constants:

`STEPPER_WAIT`	– Function will BLOCK until the stepper motion completes.
`STEPPER_NO_WAIT`	– Function will set the parameters, initiate the motion and move on.

`pNotifyFlag` – You must pass to this argument the *address of* a structure of type `STEPPER_NOTIFY`. This structure contains two fields that are set to 1 to indicate that the motion for either the left, right, or both motors has completed. This parameter is only useful if `wait_mode` is specified with `STEPPER_NO_WAIT` – otherwise, you should specify `NULL` for this parameter.

The `STEPPER_NOTIFY` structure has the following form:

```
typedef volatile struct STEPPER_NOTIFY_TYPE {

    STEPPER_FLAG left;
    STEPPER_FLAG right;

} STEPPER_NOTIFY;
```

IF you supply the address of a structure to this parameter, then after `STEPPER_step()` is issued it will immediately reset the *left* and *right* fields in this structure to **0**. The function will then – upon completion of a stepper motion – set the corresponding fields to **1** to let you know that the motion has completed. See the 'Example' section to give you an idea on how you might use this feature.

Example:

The following example does the same thing as `STEPPER_run()`, but now in *step* mode:

```
// Assume the STEPPER module has been properly opened.

// Set the acceleration to 400 steps/sec^2 -- just for fun.
STEPPER_set_accel( STEPPER_BOTH, 400 );

// GO!
STEPPER_step(  STEPPER_BOTH,              // Affect both steppers.
               STEPPER_FWD,        // Move them forward.
               600,                // Distance = 600 steps (3 revolutions).
               150,                // @ 150 steps/sec.
               STEPPER_BRK_OFF,    // Keep brakes OFF when the motion completes.
               STEPPER_WAIT,              // Wait 'HERE' until the motion
completes.
               NULL );                    // Not applicable here.
```

The follow example how you might make use of the *notify* feature (see next page):

```
// Assume the STEPPER module has been properly opened.

STEPPER_NOTIFY motion_done;

// Set the acceleration to 400 steps/sec^2 -- just for fun.
STEPPER_set_accel( STEPPER_BOTH, 400 );

// GO!
STEPPER_step(  STEPPER_BOTH,              // Affect both steppers.
               STEPPER_FWD,         // Move them forward.
               600,                 // Distance = 600 steps (3 revolutions).
               150,                 // @ 150 steps/sec.
               STEPPER_BRK_OFF,     // Keep brakes OFF when the motion completes.
               STEPPER_NO_WAIT,     // Don't wait... just GO!
               &motion_done );      // Not applicable here.

// Do other stuff right away!...
foo();
bar();
etc();

// But let's wait here.... until the motion completes.
while( ( !motion_done.left ) || ( !motion_done.right ) );

// After that, continue doing whatever...
```

The `STEPPER_stepwt()` *Macro-Function*

Format:

`STEPPER_stepwt(` *`which`*`,` *`dir`*`,` *`nSteps`*`,` *`nStepsPerSecond`*`,` *`onStopDoWhat`* `)`

Description:

This *macro-function* is a simplified version of the `STEPPER_step()` function previously discussed with the benefit that you only specify a lesser number of parameters while internally supplying `NULL` to the `pNotifyFlag`, and automatically setting the wait mode to '`STEPPER_WAIT`'. Therefore, invoking this macro-function will BLOCK until the motion completes.

Input Arguments:

This is a *wrapper macro-function* around `STEPPER_step()`. See the 'Input Arguments' section for that function.

Example:

Same example as that given for `STEPPER_step()` above. Notice the missing parameters (*see next page*):

```
// Assume the STEPPER module has been properly opened.

// Set the acceleration to 400 steps/sec^2 -- just for fun.
STEPPER_set_accel( STEPPER_BOTH, 400 );

// GO! and WAIT until the motion completes!
STEPPER_stepwt(      STEPPER_BOTH,              // Affect both steppers.
                STEPPER_FWD,        // Move them forward.
                600,                // Distance = 600 steps (3 revolutions).
                150,                // @ 150 steps/sec.
                STEPPER_BRK_OFF );  // Keep brakes OFF when the motion completes.
```

You can use this function to create complex motions from fundamental finite motions as so:

```
void avoid_obstacle( void )
{

  // Back up.
  STEPPER_stepwt( STEPPER_BOTH, STEPPER_REV, 600, 150, STEPPER_BRK_OFF );

  // Funny turn.
  STEPPER_stepwt( STEPPER_LEFT, STEPPER_FWD, 600, 150, STEPPER_BRK_OFF );

  // Move Forward a bit.
  STEPPER_stepwt( STEPPER_BOTH, STEPPER_FWD, 600, 150, STEPPER_BRK_OFF );

  // Funny turn.
  STEPPER_stepwt( STEPPER_RIGHT,STEPPER_FWD, 600, 150, STEPPER_BRK_OFF );

  // Move forward again.
  STEPPER_stepwt( STEPPER_BOTH, STEPPER_FWD, 600, 150, STEPPER_BRK_OFF );

} // end avoid_obstacle()
```

Because you're invoking the BLOCKING version of the stepping function, each function is guaranteed to execute in an orderly fashion through completion.

The `STEPPER_stepnb()` *Macro-Function*

Format:

`STEPPER_stepwt(` *`which`*, *`dir`*, *`nSteps`*, *`nStepsPerSecond`*, *`onStopDoWhat`* `)`

Description:

This *macro-function* is also a simplified version of the `STEPPER_step()` function. It takes a smaller number of parameters just like `STEPPER_stepnb()` does. In fact, it takes the same exact parameters as `STEPER_stepnb()` with the difference that the wait mode is internally set to `STEPPER_NO_WAIT` and the `pNotifyFlag` is passed the address of an *internal* notification structure that you don't get access to. This function will NOT BLOCK, and it moves on to the next instruction as soon as all the parameters are set and the motors are in motion. It is useful in circumstances where this kind of behavior is needed – such as in a *while* loop.

(*Continued on next page*)

(*Continued from previous page*)

Since there's no way to supply the address of your own notification structure like you can with `STEPPER_step()`, you have to apply a different technique to wait on motion completion to control execution. To do this, you must use `STEPPER_wait_on()` or `STEPPER_wait_and_then()` functions. See the info for *those* functions for additional details. The example below gives some ideas.

Input Arguments:

This is a *wrapper macro-function* around `STEPPER_step()`. See the 'Input Arguments' section for that function.

Example:

Here's how you might use this function:

```
// Assume the STEPPER module has been properly opened.

// Set the acceleration to 400 steps/sec^2 -- just for fun.
STEPPER_set_accel( STEPPER_BOTH, 400 );

// GO! - Issue instruction and DONT'T WAIT! (DON'T BLOCK EXECUTION).
STEPPER_stepnb(      STEPPER_BOTH,              // Affect both steppers.
                 STEPPER_FWD,        // Move them forward.
                 600,                // Distance = 600 steps (3 revolutions).
                 150,                // @ 150 steps/sec.
                 STEPPER_BRK_OFF );  // Keep brakes OFF when the motion completes.

// Do other stuff right away...
foo();
bar();
etc();

// But let's wait here.... until the motion completes for BOTH motors.
STEPPER_wait_on( STEPPER_BOTH );

// After that, continue doing whatever...
```

The `STEPPER_move()` Function

Format:

```
void STEPPER_move( STEPPER_RUNMODE      run_mode,
                   STEPPER_ID           which,

                   STEPPER_DIR          dir_L,
                   unsigned short int   steps_L,
                   unsigned short int   speed_L,
                   unsigned short int   accel_L,
                   STEPPER_BRKMODE      brkmode_L,
                   STEPPER_EVENT_PTR    step_event_L,

                   STEPPER_DIR          dir_R,
                   unsigned short int   steps_R,
                   unsigned short int   speed_R,
                   unsigned short int   accel_R,
                   STEPPER_BRKMODE      brkmode_R,
                   STEPPER_EVENT_PTR    step_event_R );
```

Description & Motivation:

By now, you may or may not have noticed the following. All of the *motion-initiating* functions we've discussed so far take on a parameter '*which*', that specifies *which* stepper is to be affected by the motion command. However, even if we can specify `STEPPER_BOTH` for the '*which*' parameter, you don't really get to specify *separate unique* settings for the left and right motor. All that happens is that the *same* parameters are applied to *both* motors.

How can you specify *independent parameters* for the left and right motors? You use the mother of all motion-initiating functions in this module – you use `STEPPER_move()`.

`STEPPER_move()` is a *hybrid* compound function of all the motion-initiating functions discussed thus far. You can specify any of the three operating *run modes*: *free-running, step (block), step (no-block)*, then you can specify the *direction*, *number of steps* (if in *step* mode), *speed*, *acceleration*, *braking* or *stop* mode, and a new parameter that allows you to specify something called a *stepper event*.

You can pretty much do everything with this one function call and it will probably be the preferred method to initiate a stepper motion.

Note that even though this function, like most functions in this chapter, takes on a '*which'* parameter. If you specify the *left* stepper, then only the corresponding parameters for the *left* stepper are used; if you specify the *right* stepper, then only the corresponding parameters for the *right* stepper are used; and if you specify *both*, then the corresponding *left* and *right* parameters are used. In any case ALL PARAMETERS ARE ALWAYS REQUIRED! Even if some become 'don't-care' as a result of only *one* stepper being affected! It is the price you pay for the convenience of a single function call.

Input Arguments:

`run_mode` – Must be one of the following enumerated constants – note these enumerated constants are NOT the same you specify to the `STEPPER_set_mode()` function:

`STEPPER_FREERUNNING` – Motors will move indefinitely.

`STEPPER_STEP_BLOCK` – Motors will move for a *finite* distance specified by '`steps_L`' and '`steps_R`' and the function will BLOCK until motions are completed.

`STEPPER_STEP_NO_BLOCK` – Motors will move for a finite distance specified by '`steps_L`' and '`steps_R`' and the function will immediately exit and move on.

`which` - Must be one of the following enumerated constants:

`STEPPER_LEFT` – Only the *left* stepper will be affected. Parameters for the *right* stepper will be ignored, but are REQUIRED.

`STEPPER_RIGHT` – Only the *right* stepper will be affected. Parameters for the *left* stepper will be ignored, but are REQUIRED.

`STEPPER_BOTH` – Both steppers will be affected.

`dir_L/dir_R` – Must be one of the following enumerated constants:

`STEPPER_FWD` – The corresponding motor will move *forward*.
`STEPPER_REV` – The corresponding motor will move in *reverse.*

`steps_L/steps_R` – This specifies the *distance* that each stepper motor is to move. The distance is given as a number of *steps* (hence the name *stepper motor*). The maximum value is **65535** steps. At the time this document was being written, there are 200 steps per revolution, so this allows a maximum motion for each stepper of approximately 327 revolutions. The value for this parameter is a "don't-care" if the specified `run_mode` is `STEPPER_FREERUNNING`.

`speed_L/speed_R` – This specifies the speed (in *steps/sec*) that the specified stepper(s) will move. At the time this document was being written the maximum value is hard-coded to **~400** steps/sec. Values higher than this will be clipped to this hard-coded value.

`accel_L/accel_R` – This specifies the acceleration (in *steps/sec*^{2}) that the specified stepper(s) will use to ramp up to the specified speeds as given by `speed_L/speed_R`. At the time this document was being written, the maximum acceleration is hard-coded to **1000** steps/sec^2. If the acceleration is set to *zero*, the acceleration feature is effectively disabled, which may be desirable under certain scenarios.

In addition, acceleration parameters are equally applied to *deceleration*. That is specifying a non-zero acceleration value will be applied equally upon deceleration. Presently, acceleration and deceleration cannot be specified as separate parameters.

`brkmode_L/brkmode_R` – This parameter must be one of the following enumerated constants and it is valid only when the *run mode* is one of the *step* modes. It is "don't-care" for *free-running*:

`STEPPER_BRK_OFF` – When motion completes for the specified motor(s), the brakes will remain disengaged.

`STEPPER_BRK_ON` – When the motion completes for the specified motor(s), the brakes will be engaged, and will remain so until the user explicitly "removes the brakes" via issuing a `STEPPER_stop()`.

> **Note:** Use *brakes* judiciously – braking can really drain your battery.

`step_event_L/step_event_R` – This parameter allows you to specify a *stepper event* function that will be invoked when motion completes, provided the *run mode* is *step-block* mode – specified via `STEPPER_STEP_BLOCK`. It is "don't-care" in other modes and you *must* pass `NULL` in such cases. Please refer to the last example in the 'Example' section for *this* function which shows how to use stepper events.

Example:

Let us look at some examples. This first example makes the CEENBoT turn indefinitely – *just* because you can! Here we invoke the *free-running* mode, which means the CEENBoT will never stop turning until you make it stop (see `STEPPER_stop()`). Note also in this example, that the number of steps is *zero*, because *distance* has no meaning in *free-running* mode. The same is true for the *brake mode*, which is also 'don't-care' in *free-running* mode since it's never going to stop!

```
// Assume the STEPPER module has been properly opened.

// Make the 'BoT turn indefinitely.
// Stepper distance = Don't Care (0); Speed = 400 steps/sec; Accel = 400 steps/sec^2
// Braking = Don't care in free-running; Stepper Events not applicable (NULL).
STEPPER_move( STEPPER_FREERUNNING, STEPPER_BOTH,

         STEPPER_FWD, 0, 200, 400, STEPPER_BRK_OFF, NULL,    // left
         STEPPER_REV, 0, 200, 400, STEPPER_BRK_OFF, NULL );  // right
```

In the second example given for `STEPPER_stepwt()`, we showed how a function for a complex motion could be created by making multiple calls to the *blocking* version of the stepping function, the *turns* were not true turns (referred in the example as '*funny turns*') because we couldn't turn one wheel forward and one in reverse at the same time (although there *is* a way to accomplish this with the elementary functions). In any case, here's the same, but *improved* example:

```
// Assume the STEPPER module has been properly opened.

void avoid_obstacle( void )
{

  // Back up!
  STEPPER_move( STEPPER_STEP_BLOCK, STEPPER_BOTH,

         STEPPER_REV, 600, 200, 400, STEPPER_BRK_OFF, NULL,   // left
         STEPPER_REV, 600, 200, 400, STEPPER_BRK_OFF, NULL ); // right

  // Turn RIGHT.
  STEPPER_move( STEPPER_STEP_BLOCK, STEPPER_BOTH,

         STEPPER_FWD, 600, 200, 400, STEPPER_BRK_OFF, NULL,   // left
         STEPPER_REV, 600, 200, 400, STEPPER_BRK_OFF, NULL ); // right


  // Move Forward a bit.
  STEPPER_move( STEPPER_STEP_BLOCK, STEPPER_BOTH,

         STEPPER_FWD, 600, 200, 400, STEPPER_BRK_OFF, NULL,   // left
         STEPPER_FWD, 600, 200, 400, STEPPER_BRK_OFF, NULL ); // right

  // Now turn LEFT.
  STEPPER_move( STEPPER_STEP_BLOCK, STEPPER_BOTH,

         STEPPER_REV, 600, 200, 400, STEPPER_BRK_OFF, NULL,   // left
         STEPPER_FWD, 600, 200, 400, STEPPER_BRK_OFF, NULL ); // right
```

(*Continued on next page*)

(*Continued from previous page*)

```
    // Now move FORWARD forever until we need to 'avoid' again.
    STEPPER_move( STEPPER_FREERUNNING, STEPPER_BOTH,

            STEPPER_FWD, 0, 200, 400, STEPPER_BRK_OFF, NULL,   // left
            STEPPER_FWD, 0, 200, 400, STEPPER_BRK_OFF, NULL ); // right

} // end avoid_obstacle()
```

The next example moves the *left* stepper motor a finite distance. Notice how *all parameters* are required, as already mentioned, even if they're considered 'don't-care'.

```
// Assume the STEPPER module has been properly opened.

// Move the left stepper a finite distance.  Don't do anything else
// until the motion completes.
STEPPER_move( STEPPER_STEP_BLOCK, STEPPER_LEFT,

         STEPPER_FWD, 600, 200, 400,  STEPPER_BRK_OFF, NULL,    // Left
         STEPPER_FWD,   0,   0,   0,  STEPPER_BRK_OFF, NULL );  // Right
```

The last example is a more comprehensive example and shows how *stepper events* can be implemented.

```
#include "capi324v221.h"

// -----------Prototypes:
STEPPER_EVENT( left_motor_done  );// Stepper event prototype.
STEPPER_EVENT( right_motor_done );// Stepper event prototype.

// -----------Stepper Event Function Implementation:

STEPPER_EVENT( left_motor_done ) {

    // Just display a message that the motor has stopped.
    LCD_printf_RC( 3, 0, "LEFT Motor Stopped!" );

} // end left_motor_done()

GPI_STEPPER_EVENT( right_motor_done ) {

    // Just display a message that the motor has stopped.
    LCD_printf_RC( 2, 0, "RIGHT Motor Stopped!" );

} // end right_motor_done()
```

(*Continued on next page*)

(*Continued from previous page*)

```
// -----------CBOT-main:

void CBOT_main( void ) {

    // Open the LCD and STEPPER subsystems - let us assume there were no errors.
    LCD_open();
    STEPPER_open();

    // Just issue a 'STEPPER_move()' and we'll just wait for when
    // each motor completes.  The left motor will move 600 steps (3 revolutions)
    // at 200 steps/sec, with acceleraton set to 300.  The right motor will move
    // 300 steps (1.5 revolutions) at 200 steps/sec with acceleraton of 100.
    // Both steppers will engage the brakes when motion completes, and in addition
    // 'left_motor_done()' will be triggered when the left stepper completes its
    // motion and 'right_motor_done()' will trigger when the right stepper completes
    // its motion.
    STEPPER_move( STEPPER_STEP_BLOCK, STEPPER_BOTH,

            STEPPER_FWD, 600, 200, 300, STEPPER_BRK_ON, left_motor_done,
            STEPPER_REV, 300, 200, 100, STEPPER_BRK_ON, right_motor_done );


    // Wait a bit for the motion to complete (5-seconds) - (See TMRSRVC chapter).
    TMRSRVC_delay( 5000 );

    // Disengage the brakes -- we don't want to drain the battery!
    STEPPER_stop( STEPPER_BOTH, STEPPER_BRK_OFF );

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

} // end CBOT_main()
```

Note: It is more likely that you may not even care about using *stepper events*. So, to keep you from always having to specify `NULL` for these parameters should should consider the following macro-functions `STEPPER_move_rn()`, `STEPPER_move_stnb()` and `STEPPER_move_stwt()`, which are slightly simpler versions of `STEPPER_move()` with fewer parameters. These macro-functions are discussed next.

The `STEPPER_move_rn()` *Macro-Function*

Format:

```
STEPPER_move_rn( which,  dir_L,  speed_L, accel_L, dir_R, speed_R, accel_R )
```

Description:

This is a wrapper around the `STEPPER_move()` function with the *run mode* automatically set to `STEPPER_FREERUNNING`, the step distance automatically set to `0`, and the *stepper event* parameters internally passed `NULL` since these parameters are not applicable to *free-running* mode. Please refer to the information given for `STEPPER_move()` for additional details.

Input Arguments:

Since this is a *wrapper* macro for `STEPPER_move()`, see the parameters for *that* function for details.

Example:

Just a quick comparison as an example – the superfluous parameters are highlighted blue:

```
// The normal way...
STEPPER_move( STEPPER_FREERUNNING, STEPPER_BOTH,

          STEPPER_FWD, 0, 200, 400, STEPPER_BRK_OFF, NULL,      // Left
          STEPPER_FWD, 0, 200, 400, STEPPER_BRK_OFF, NULL );   // Right


// A BETTER way...
STEPPER_move_rn( STEPPER_BOTH,
                    STEPPER_FWD, 200, 400,    // Left
                    STEPPER_FWD, 200, 400 );  // Right
```

The `STEPPER_move_stnb()` *Macro-Function*

Format:

```
STEPPER_move_stnb( which, dir_L, steps_L, speed_L, accel_L, brkmode_L,
                          dir_R, steps_R, speed_R, accel_R, brkmode_R )
```

Description:

This is wrapper around the `STEPPER_move()` function with the *run mode* automatically set to `STEPPER_STEP_NO_BLOCK`, and the *stepper event* parameters internally passed `NULL`. Please refer to the information given for `STEPPER_move()` for additional details.

Input Arguments:

Since this is a *wrapper* macro for `STEPPER_move()`, see the parameters for *that* function for details.

(*Continued on next page*)

(*Continued from previous page*)

Example:

Just a quick comparison as an example – the superfluous parameters are highlighted blue:

```
// The normal way...
STEPPER_move( STEPPER_STEP_NO_BLOCK, STEPPER_BOTH,

         STEPPER_FWD, 1200, 200, 400, STEPPER_BRK_OFF, NULL,         // Left
         STEPPER_FWD, 1200, 200, 400, STEPPER_BRK_OFF, NULL );       // Right


// A [SLIGHTLY] BETTER way...
STEPPER_move_stnb( STEPPER_BOTH,
                        STEPPER_FWD, 1200, 200, 400, STEPPER_BRK_OFF,
  // Left
                        STEPPER_FWD, 1200, 200, 400, STEPPER_BRK_OFF );
  // Right
```

The `STEPPER_move_stwt()` *Macro-Function*

Format:

```
STEPPER_move_stwt( which, dir_L, steps_L, speed_L, accel_L, brkmode_L,
                          dir_R, steps_R, speed_R, accel_R, brkmode_R )
```

Description:

This is wrapper around the `STEPPER_move()` function with the *run mode* automatically set to `STEPPER_STEP_BLOCK`, and the *stepper event* parameters internally passed `NULL`. Please refer to the information given for `STEPPER_move()` for additional details.

Input Arguments:

Since this is a *wrapper* macro for `STEPPER_move()`, see the parameters for *that* function for details.

Example:

Just a quick comparison as an example – the superfluous parameters are highlighted blue:

```
// The normal way...
STEPPER_move( STEPPER_STEP_BLOCK, STEPPER_BOTH,

         STEPPER_FWD, 1200, 200, 400, STEPPER_BRK_OFF, NULL,         // Left
         STEPPER_FWD, 1200, 200, 400, STEPPER_BRK_OFF, NULL );       // Right


// A [SLIGHTLY] BETTER way...
STEPPER_move_stwt( STEPPER_BOTH,
                        STEPPER_FWD, 1200, 200, 400, STEPPER_BRK_OFF,
  // Left
                        STEPPER_FWD, 1200, 200, 400, STEPPER_BRK_OFF );
  // Right
```

The `STEPPER_stop()` Function

Format:

```
void STEPPER_stop( STEPPER_ID which, STEPPER_BRKMODE brakeMode )
```

Description:

This function can be used to *cancel* and/or *halt* a motion that is currently in progress regardless of the current operating *run mode*. It can also be used to engage and disengage the brakes. The exact behavior of `STEPPER_stop()` depends on the operating *run mode* at which it's issued, and whether a request to engage the brakes or not is also issued with the call.

If the operating *run mode* is *free-running*, then invoking `STEPPER_stop()` with the brakes OFF will *cancel* the motion. It will simply be the same as setting the speed to zero. Motion will stop until you initiate a *new* motion by invoking one of the already-discussed motion-initiating functions. However, If `STEPPER_stop()` is invoked with the brakes ON, then the motion is *halted*, which is not to be confused with *canceled*. That is, the speed setting will be retained until you either release the brakes by invoking `STEPPER_stop()` with the brakes OFF (at which point the motion will move from being *halted* to being *canceled*), or re-initiate the motion already in progress (based on the current settings) via `STEPPER_go()`.

(*Continued from previous page*)

Now, if the operating *run mode* is one of the *step* modes instead, then invoking `STEPPER_stop()` with the brakes OFF will also *cancel* the motion. In this mode this would be the same as setting the speed *and* the number of steps remaining to zero. Motion will stop until you re-issue a *new* motion by way of the already-discussed motion-initiating functions. However, if `STEPPER_stop()` is invoked with the brakes ON, then the motion is *halted*, which is not to be confused with *canceled*. That is, the speed and number of remaining steps will be retained until you either release the brakes by invoking `STEPPER_stop()` with the brakes OFF (at which point the motion will move from being *halted* to being *canceled*), or re-initiating the motion already in progress (based on the current settings) via `STEPPER_go()`.

Note that in *step* mode, issuing `STEPPER_go()` will only work if there are 'steps' remaining to advance before the brakes were applied. If this value is exhausted, then `STEPPER_go()` will not produce any motion. In this case, you would have to re-issue a new motion-initiating function call.

Input Arguments:

`which` – Must be one of the following enumerated constants that specified *which* stepper is to be affected by the 'set' operation:

`STEPPER_LEFT`	(or `LEFT_STEPPER`)	– Only the *left* stepper will be affected.
`STEPPER_RIGHT`	(or `RIGHT_STEPPER`)	– Only the *right* stepper will be affected.
`STEPPER_BOTH`	(or `BOTH_STEPPERS`)	– Both steppers will be affected.

`brakeMode` – You must specify one of the following enumerated constants:

`STEPPER_BRK_OFF`	– The brakes will disengage, *if* engaged.
`STEPPER_BRK_ON`	– The brakes will engage, *if* not already so.

Example:

Checkout info on `STEPPER_go()` and look up the example for *that*.

The `STEPPER_go()` Function

Format:

```
void STEPPER_go( STEPPER_ID which )
```

Description:

This function is specifically meant to be used after a previously issued call to `STEPPER_stop()` with the brakes ON. Typically, when you issue `STEPPER_stop()` with the brakes ON, the motion will be *halted*, but NOT *canceled*. Essentially, the function removes (disengages) the brakes and if in *free-running* mode, the motion will continue, or if in *step* mode, the motion will continue provided there are 'steps' remaining to advance through.

You may be asking yourself, "*but can't I disengage the brakes with* `STEPPER_stop()`*?"* – and the answer to that is YES you can, however calling `STEPPER_stop()` to disengage the brakes also *cancels* the motion if there is motion in progress that is yet to complete. Calling `STEPPER_go()`, disengages the brakes without *canceling* the motion – that is, the motion continues where it left off – again, provided there is motion that remains to be completed.

(*Continued on next page*)

(*Continued from previous page*)

Consequently the following over-simplified example with most parameters avoided for simplicity:

```
STEPPER_move( ...parameters... );      // Or ANY other motion-initiating function.

STEPPER_stop( ..., STEPPER_BRK_ON );   // Engage brakes.

// ... some time passes ...

STEPPER_stop( ..., STEPPER_BRK_OFF );  // Cancel the motion.
```

will result in the motion being ultimately *canceled*, while:

```
STEPPER_move( ...parameters... );      // Or ANY other motion-initiating function.

STEPPER_stop( ..., STEPPER_BRK_ON );   // Engage the brakes.

// ... some time passes ...

STEPPER_go( ... );                     // Continue.
```

will result in the motion continuing where it left off.

Simple as that.

Input Arguments:

`which` – Must be one of the following enumerated constants that specifies *which* stepper is to be affected by the this operation:

`STEPPER_LEFT`	(or `LEFT_STEPPER`)	– Only the *left* stepper will be affected.
`STEPPER_RIGHT`	(or `RIGHT_STEPPER`)	– Only the *right* stepper will be affected.
`STEPPER_BOTH`	(or `BOTH_STEPPERS`)	– Both steppers will be affected.

Example:

A more concrete 'snippet':

```
// Assume the STEPPER module has been properly opened.

// Move the steppers for several revolutions... say 6 revolutions = 1200 steps.
// Speed = 200; No Acceleration; Brakes remain OFF; No Stepper Events.
STEPPER_move( STEPPER_STEP_NO_BLOCK, STEPPER_BOTH,

  STEPPER_FWD, 1200, 200, 0,   STEPPER_BRK_OFF, NULL,       // Left
  STEPPER_FWD, 1200, 200, 0,   STEPPER_BRK_OFF, NULL  );    // Right


// Wait 2 seconds... there's no way the motion will finish in this time.
TMRSRVC_delay( TMR_SECS( 2 ) );
```

(*Continued on next page*)

(*Continued from previous page*)

```
// Enage the brakes and 'halt' the motion.
STEPPER_stop( STEPPER_BOTH, STEPPER_BRK_ON );

// Wait 2 seconds...
TMRSRVC_delay( TMR_SECS( 2 ) );

// Continue.
STEPPER_go( STEPPER_BOTH );

// Wait for all motion to complete.
```

Function Reference: *Wait* Functions

When operating the steppers with the *run mode* set to the *non-blocking step* modes, it may become necessary to prevent '*runaway program execution*'. What does this mean exactly? Recall that invoking a motion-initiating function in one of the *non-blocking step* modes will result in the function starting up the motion and exiting as soon as possible, whereby any other following functions or expression will immediately execute while the motion is still in effect.

There may be times, however, where you want to start a motion, and immediately do something as soon as you call the motion-initiating function, but then, after a few instructions or function calls, you may want to wait on the motion you initiated to complete before you do anything else. You need '*controlled program execution*'.

The *wait* functions are precisely used for this purpose. To control program execution when you're taking advantage of *non-blocking* functions. The 'wait' functions only work when you're operating in *step* mode, because recall that step mode motion is finite – it is bound to end at some point. Waiting on *free-running* mode will have you waiting forever and hence why things are the way they are.

So, *wait* functions can be used specifically after any of the following motion-initiating functions:

- `STEPPER_stepnb()`
- `STEPPER_move()` (when *run mode* is `STEPPER_STEP_NO_BLOCK` only)
- `STEPPER_move_stnb()`

Note that these are all *non-blocking* functions. It doesn't make sense to wait on a *blocking* function, as the behavior of the blocking function is itself 'waiting' for motion to complete.

> **Note:** If you try to use a *wait* function with a motion-initiating function other than what's listed above you will be waiting indefinitely. Your program will *freeze*!

Let's get to it.

The `STEPPER_wait_on()` Function

Format:

```
void STEPPER_wait_on( STEPPER_ID which )
```

Description:

This function can be used to 'wait' on a stepper motion to complete. Program execution will hold at `STEPPER_wait()` until the specified motor(s) complete their motion.

Input Arguments:

`which` – Must be one of the following enumerated constants that specifies *which* stepper is to be affected by the this operation – that is, the stepper you want to 'wait' on before moving on with program execution:

`STEPPER_LEFT`	(or `LEFT_STEPPER`)	– Wait on the *left* stepper to complete its motion.
`STEPPER_RIGHT`	(or `RIGHT_STEPPER`)	– Wait on the *right* stepper to complete its motion.
`STEPPER_BOTH`	(or `BOTH_STEPPERS`)	– Wait on both steppers to complete their motion.

(*Continued from previous page*)

Example:

```
// Assume the STEPPER module has been properly opened.

// Assume the LCD module has been properly opened.

// Initiate a non-blocking STEP motion -- RIGHT stepper only.
// Forward motion; Distance = 1200 steps; Speed = 200 steps/sec;
// Engage brakes when motion completes.
STEPPER_stepnb( STEPPER_RIGHT, STEPPER_FWD, 1200, 200,  STEPPER_BRK_ON );

// ... meanwhile... while we wait on that to complete...
LCD_clear();
LCD_printf( "Look, Ma!\n" );
LCD_printf( "Wheels are rollin'!\n" );
LCD_printf( "What'cha think?\n" );

// Let's hold off HERE and wait before we do anything else.
STEPPER_wait_on( STEPPER_RIGHT );

LCD_clear();
LCD_printf( "Motion complete!\n" );
LCD_printf( "Smooth operator.\n" );
LCD_printf( "     ;)       \n" );
```

The `STEPPER_wait_and_then()` Function

Format:

```
void STEPPER_wait_and_then( STEPPER_ID which, ... )
```

Description:

This function works *just* like `STEPPER_wait_on()`, but in addition, you can supply one or two additional parameters, being the name of your *stepper events*, provided that you have defined them. The function will then *wait* on the specified stepper motor(s) to complete their motion, at which point, your *stepper event* functions will be triggered – provided you have supplied one.

Input Arguments:

`which` – Must be one of the following enumerated constants that specifies *which* stepper is to be affected by the this operation – that is, the stepper you want to 'wait' on before moving on with program execution:

`STEPPER_LEFT`	(or `LEFT_STEPPER`)	– Wait on the *left* stepper to complete its motion.
`STEPPER_RIGHT`	(or `RIGHT_STEPPER`)	– Wait on the *right* stepper to complete its motion.
`STEPPER_BOTH`	(or `BOTH_STEPPERS`)	– Wait on both steppers to complete their motion.

`...` - You *must* supply at least ONE more parameter, depending on what you specified for `which` above. If you specified either *left* or *right* steppers (referring to *just* one), then you can pass as a parameter, either the name of a *stepper event* (if you have declared and defined one in your program), or `NULL`. However, if you specified *both* steppers, then you *must* specify TWO parameters, the name of a *stepper event* for the LEFT motor (or `NULL`), and the name of the *stepper event* for the RIGHT motor (or `NULL`) – in *that* order.

Example:

```
#include "capi324v221.h"

// ------------ prototypes:
STEPPER_EVENT( evt_left_stepper  );
STEPPER_EVENT( evt_right_stepper );

// ------------ functions:
STEPPER_EVENT( evt_left_stepper ) {

  // Print at the bottom of the LCD display.
  LCD_printf_RC( 0, 0, "Left complete!" );

  // ... your short-and-sweet code here ...

} // end evt_left_stepper()

STEPPER_EVENT( evt_right_stepper ) {

  // Print at second to last line (bottom) of LCD display.
  LCD_printf_RC( 1, 0, "Right complete!" );

  // ... your short-and-sweet code here ...

} // end evt_right_stepper()

// -------------- CBoT main:

void CBOT_main( void ) {

  LCD_open();           // Open the LCD.
  STEPPER_open();       // Open the STEPPER.

  // Initiate a motion.
  STEPPER_move( STEPPER_STEP_NO_BLOCK, STEPPER_BOTH,

         STEPPER_FWD, 1200, 200, 400, STEPPER_BRK_OFF, NULL,
         STEPPER_FWD, 1200, 200, 400, STEPPER_BRK_OFF, NULL  );

  // While that's happening...
  LCD_printf( "CEENBoT Pimpin'!\n" );
  LCD_printf( "What's up!?\n" );

  foo();
  bar();

  // Okay, let's HOLD **HERE** and trigger the specified stepper events
  // once motion completes!
  STEPPER_wait_and_then( STEPPER_BOTH, evt_left_stepper, evt_right_stepper
);

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

} // end CBOT_main()
```

Chapter 11: The SWATCH (Stopwatch)

This chapter introduces you to the functional services offered by the **`SWATCH`** or *stopwatch* subsystem module and how it can be used to measure small-scale time.

Module at a Glance

Description

The **`SWATCH`** subsystem module (also referred to as the **`STOPWATCH`** module) allows users to perform *'small-scale'* time measurements with a minimum granularity of 10 *microseconds* per *tick*. Essentially, once the **`SWATCH`** module is opened, you START the stopwatch, perform some action or execute some functions that happen very quickly and then STOP it (the stopwatch) and request the time that has elapsed, which is given as a number of *ticks*. You then multiply this value by **`10`** to obtain the time elapsed in units of microseconds.

The primary motivation for the existence of this module is to measure the 'echo' time obtained from the *ultrasonic* device that can be attached to the CEENBoT. Consequently, the ultrasonic module (**`USONIC`**) requires the stopwatch service to be opened for **`USONIC`** to operate correctly since it relies on the stopwatch service.

In any case, you may use the stopwatch service for your own needs.

Presently, this feature is somewhat limited. The stopwatch can only perform measurements from **`0us`** up to **`655.35ms`** (or **`655350us`**). This limitation is due to the fact that the stopwatch service is implemented by using the 16-bit timer (**`Timer1`**) on the MCU. This limitation may change in the future to include larger time measurements (*large-scale*).

Modular Dependencies

The SWATCH module *must* be manually opened by the user. It has no other modular dependencies.

Hardware Dependencies

The following MCU-specific dependencies exist:

- **`Timer1`** – The **`SWATCH`** module is entirely driven by this timer.
- NO *interrupt service routines* (ISRs) related to **`Timer1`** may be used nor invoked once the stopwatch service has been started, whether through the API (by way of `CBOT_ISR()`), or outside of it by bypassing it (e.g., by using the `ISR()` macro).

> **Note:** The *second* condition regarding ISRs above is only applicable when you open the **`SWATCH`** module for use. If you do NOT open the module, both **`Timer1`** and corresponding ISRs are available for your *custom* use. Note that once you open the **`SWATCH`** module by invoking `STOPWATCH_open()` somewhere in your code, you've already paid the price regarding the 'highjacking' of **`Timer1`** and any corresponding ISRs, even if you eventually *close* it after you're done with it. So, you either choose to use the **`SWATCH`** service in your program (which 'hogs' **`Timer1`**), or NOT at all.
>
> Consequently, *closing* the **`SWATCH`** module after you're done with it *will* release **`Timer1`** back to you, but it will not release the ISRs that have been 'highjacked' by the API, which means you won't be able to write your own corresponding **`Timer1`** related ISRs.
>
> It is hoped that this limitation will be resolved in future revisions of the API.

(*Continued on next page*)

(*Continued from previous page*)

- The `PCINT0_vect` ISR is also not available to the user when this module is started. The user should not attempt to write his/her own ISR routine around this particular interrupt vector as it is 'highjacked' by the API.

Additional Restrictions to Consider (as of API Rev. v1.02.000R)

As of API revision **`1.02.000R`** (and higher), the **`SPKR`** subsystem module has now been implemented that allows users to generate sounds and musical notes. The tone synthesis mechanism also relies on **`Timer1`**. Consequently, this means that **`Timer1`** is shared between these two subsystem modules (the **`STOPWATCH`**, and the **`SPKR`**), so these two subsystem modules **CANNOT BOTH BE OPEN AT THE SAME TIME**. Therefore, if you have the **`STOPWATCH`** service running (open), you CANNOT open the **`SPKR`** subsystem module in *tone* mode. The reverse is also true: if the **`SPKR`** subsystem module is currently open, then you cannot open the **`STOPWATCH`** service. Check out the chapter on the **`SPKR`** subsystem module for more details.

Just keep that in mind.

Function List Summary

- `STOPWATCH_open()` - Opens the **`STOPWATCH`** service subsystem module.
- `STOPWATCH_close()` - Close the **`STOPWATCH`** service subsystem module.
- `STOPWATCH_start()` - Starts and resets the stopwatch (starts *counting*).
- `STOPWATCH_stop()` - Stops the stopwatch and returns the elapsed time, in *ticks*.
- `STOPWATCH_reset()` - Resets the stopwatch counter to zero.
- `STOPWATCH_set()` - Sets an initial count value in *ticks* before starting.
- `STOPWATCH_get_ticks()` - Gets the current stopwatch time, in *ticks*.

> **Note:** The remainder of this chapter uses the term **`SWATCH`** and **`STOPWATCH`** to refer to the same subsystem module. The terms are used interchangeably.

Function Reference

The `STOPWATCH_open()` Function

Format:

```
SUBSYS_OPENSTAT STOPWATCH_open( void )
```

Description:

Function acquires and initializes resources needed for operation of the **SWATCH** subsystem module. You *must* call this function first with a successful 'open' before invoking any other function provided by this module.

Returns:

Returns a structure of type `SUBSYS_OPENSTAT` whose field entries indicate the status of the open request and the subsystem that resulted in an error (when, and *if* and error occurs). See *Chapter 1, Procedure for Opening Modules Before Use* for examples on how to handle the *open status*.

Example:

```
#include "capi324v221.h"

void CBOT_main( void )
{

  SUBSYS_OPENSTAT opstat;

  // Open the STOPWATCH module.
  opstat = STOPWATCH_open();

  if ( opstat.state == SUBSYS_OPEN )
  {

         // ... DO STOPWATCH STUFF ...

  }// end if()

} // end CBOT_main()
```

The `STOPWATCH_close()` Function

Format:

```
void STOPWATCH_close( void )
```

Description:

Function deallocates and releases resources being used by the **STOPWATCH** subsystem module. No other functions should be invoked once the subsystem module is closed.

The `STOPWATCH_start()` Function

Format:

```
void STOPWATCH_start( void )
```

Description:

Function triggers the stopwatch to START *counting*. Note that the counting begins from whatever the current value is in the stopwatch's internal counting register. To start counting from *zero*, you should *reset* the counter first via `STOPWATCH_reset()`.

Example:

See the example for `STOPWATCH_stop()`.

The `STOPWATCH_stop()` Function

Format:

```
SWTIME STOPWATCH_stop( void )
```

Description:

Function triggers the stopwatch to STOP *counting* after a prior call to `STOPWATCH_start()`.

Returns:

Function returns a value of type `SWTIME` – which is essentially an `unsigned short int`. This value represents the number of *ticks* elapsed since the stopwatch was started from its current value. You must multiply this value by `10` to find the time elapsed in *microseconds*, since each *tick* is worth **`10us`**.

Example:

```
// Assume the STOPWATCH service is properly opened.

// Assume the LCD module is properly opened.

SWTIME sw_time_ticks;          // Store number of 'ticks'.

unsigned long int time_us;     // Store time in us.

// Start from zero.
STOPWATCH_reset();

// Start the stopwatch.
STOPWATCH_start();

// ... do stuff quickly ...

foo();
bar();
```

(*Continued on next page*)

(*Continued from previous page*)

```
// Find out how long that took.
sw_time_ticks = STOPWATCH_stop();

// Convert to microseconds.  (Note we need to type-cast).
time_us = 10 * ( ( unsigned long int )( sw_time_ticks ) );

// Show the result.
LCD_printf( "Time elapsed:\n"  );
LCD_Printf( " = %d\n", time_us );
```

Note, in the above example, how `time_us` was declared as `unsigned long int`, which provides *larger* storage than `SWTIME` (which is an `unsigned short int`). This is to prevent *overflow* after we multiply the number of ticks by 10, which might exceed the maximum capacity of `SWTIME`. This is also the reason behind the need to *type-cast* from the smaller storage type to the larger one as we convert from *ticks* to *microseconds*.

The `STOPWATCH_reset()` Function

Format:

```
SWTIME STOPWATCH_reset( void )
```

Description:

Function resets the internal stopwatch counter to zero for a clean stopwatch measurement start.

Returns:

As a convenience the function returns a value of type **`SWTIME`** consisting of whatever the current *count* value (in *ticks*) was before resetting the internal count register. You must multiply this value by **`10`** to convert the time to units of *microseconds*.

Example:

See the example for `STOPWATCH_stop()`.

The `STOPWATCH_set()` Function

Format:

```
void STOPWATCH_set( SWTIME value )
```

Description:

Function allows the user to set the *initial count* (in *ticks*) to begin with, before issuing a `STOPWATCH_start()`.

Input Arguments:

A value of type `SWTIME`, which is essentially an `unsigned short int`. So this value is restricted to **`0`** to **`65535`**. Note that if the value you're supplying is in units of *microseconds*, then you have to divide by **`10`** to convert to units of *ticks*.

The `STOPWATCH_get_ticks()` Function

<u>Format</u>:

```
SWTIME STOPWATCH_get_ticks( void )
```

<u>Description</u>:

Function allows the user to get the current stopwatch count value without having to stop the stopwatch to find out what this value is.

<u>Returns</u>:

Function returns a value of type `SWTIME`, which is essentially an `unsigned short int`. This value represents the current count value (in *ticks*) since the stopwatch started. You must multiply this value by **`10`** to get the time representation in *microseconds*.

Chapter 12: The TINY Subsystem Module

> The **`TINY`** subsystem module provides functions for communicating with the *secondary* supporting MCU on the CEENBoT, which happens to be in command of *switch* and *IR sensor* state, as well as *RC servo* control.

Module at a Glance

Description

The **`TINY`** subsystem module provides a set of functions that allow you to communicate with the *secondary* supporting MCU on the CEENBoT. Presently, this secondary MCU is the **`ATtiny48`** and is the reason behind the name '**`TINY`**' assigned for this particular module. The secondary MCU is in control of the following on-board features:

- State of the push-button *switches*
- State of the IR (Infra-Red) *sensors*
- Control of the RC Servos (when attached)

Therefore, in order for you to make use of any of these aforementioned features, you have to go *through* the 'Tiny, and the reason these 'services' are collectively grouped under this particular module. So let's get to it.

Modular Dependencies

The **`TINY`** module is already *open by default* since it constitutes a critical component for the CEENBoT. In addition, it has the following *modular dependencies*:

- **`SPI`** - The **`ATtiny`** is an SPI device (from the point of view of the *primary* MCU).

> **Note:** The **`SPI`** is also *open by default*.

Hardware Dependencies

The **`TINY`** module has *no* hardware dependencies on the *primary* MCU.

Function List Summary

- `ATTINY_open()` - Open the TINY module for use.
- `ATTINY_close()` - Close the TINY module.
- `ATTINY_get_firm_rev()` - Obtain the *firmware* revision running on the Attiny.

- `ATTINY_get_sensors()` - Get *switch* and *IR Sensor* status.
[***Notice: Usage of the `ATTINY_get_sensors()` function has been superseded by `ATTINY_get_IR_state()` and `ATTINY_get_SW_state()` functions given below. Usage of this function is *highly* discouraged.]

- `ATTINY_set_RC_servos()` - Control positioning of the RC servos.
- `ATTINY_set_RC_servo()` - Control positioning of a specific RC servo.
- `ATTINY_set_RC_limits()` - Allows you to set 'safe-guard' limits for each of the supported servos.
- `ATTINY_get_SW_state()` - Get *switch* status for a specific switch.
- `ATTINY_get_IR_state()` - Get *IR* sensor status for a specific IR.

Function Reference

The `ATTINY_open()` Function

Format:

```
SUBSYS_OPENSTAT ATTINY_open( void )
```

Description:

Function acquires and initializes resources needed for operation of the **ATTINY** subsystem module. You *must* call this function first with a successful 'open' before invoking any other function provided by this module.

Returns:

Returns a structure of type SUBSYS_OPENSTAT whose field entries indicate the status of the open request and the subsystem that resulted in an error (when, and *if* an error occurs). See *Chapter 1, Procedure for Opening Modules Before Use* for examples on how to handle the *open status*.

> **Note:** Please note that the **TINY** module is open by default.

The `ATTINY_close()` Function

Format:

```
void ATTINY_close( void )
```

Description:

Function deallocates and releases resources being used by the TINY subsystem module. No other functions should be invoked once the subsystem module is closed.

> **Note:** You shouldn't attempt to *close* the **TINY** module – it is a *critical* component of the API.

The `ATTINY_get_firm_rev()` Function

Format:

```
void ATTINY_get_firm_rev( ATTINY_FIRMREV *pDest )
```

Description:

Use this function to *poll* the 'Tiny for the current revision of its *firmware*.

Input Arguments:

`pDest` – You pass to this argument the *address of* a structure of type `ATTINY_FIRMREV`. The function will populate the field entries of the structure with the relevant revision values. The structure has the following form:

```
typedef struct ATTINY_FIRMREV_TYPE
{

    unsigned char major;        // Holds 'major' revision number.
    unsigned char minor;        // Holds 'minor' revision number.
    unsigned char status;       // Holds 'status' revision code.

} ATTINY_FIRMREV;
```

The fields `major` and `minor` are treated as *integer* values, where as `status` is treated as a *character*.

Example:

```
// Assume the TINY module is properly opened.

// Assume the LCD module is properly opened.

ATTINY_FIRMREV tiny_rev;

// Get the revision running on the secondary MCU.
ATTINY_get_firm_rev( &tiny_rev );

// Print the results.
LCD_printf( "Tiny Rev: v%d.%d%c\n",    tiny_rev.major,
                                       tiny_rev.minor,
                                       tiny_rev.status );
```

The `ATTINY_get_sensors()` Function

[***Notice: Usage of the `ATTINY_get_sensors()` function has been superseded by `ATTINY_get_IR_state()` and `ATTINY_get_SW_state()` functions given below. Usage of this function is *highly* discouraged.]

Format:

```
unsigned char ATTINY_get_sensors( void )
```

Description:

This *low-level* function allows you to request *switch* and IR-sensor status information. Usage of this function is highly discouraged because `ATTINY_get_sensors()` is NOT deemed *loop-safe*. (*Continued on next page*)

(*Continued from previous page*)

As of API version `v1.02.000R`, **users should request such information via the new *loop-safe* functions `ATTINY_get_SW_state()` and `ATTINY_get_IR_state()` respectively**. Refer to these functions for additional information.

Returns:

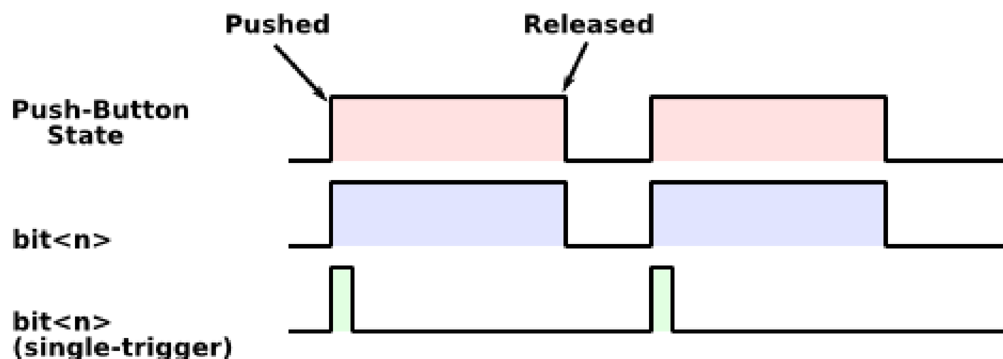
Function returns an 8-bit value as an `unsigned char`. This value has the following bit assignments:

```
bit0: Left  IR sensor state   ;1 = Active, 0 = Inactive (no obtrusion).
bit1: Right IR sensor state   ;1 = Active, 0 = Inactive (no obtrusion).
bit2: Switch state (S5)       ;1 = Depressed, 0 = Released
bit3: Switch state (S4)       ;1 = Depressed, 0 = Released
bit4: Switch state (S3)       ;1 = Depressed, 0 = Released
bit5: Switch state (S5)       ;(positive edge-triggered version)
bit6: Switch state (S4)       ;(positive edge-triggered version)
bit7: Switch state (S3)       ;(positive edge-triggered version)
```

`Bits 0-1` are straight forward – if **`1`**, that means the proximity IR sensors are active due to object obtrusion of some sort.

`Bits 2-4` are also straight forward. If the switch is *depressed* (pushed-down), then the corresponding bit position will read as **`1`** as long as the user keeps the button pushed down. It goes back to **`0`** once the user releases it.

`Bits 5-7` ALSO mirror the same switches (**`S3`** to **`S5`**) – however these are *edge-triggered* versions of the same. That is, the corresponding bits will read **`'1'`** when the user pushes the button (which results in a **`0-to-1`** transition or *positive-edge*) and then it will immediately go back to zero even after the button remains depressed and it will remain zero until the user both releases the push-button and depresses it again. This *behavior* is illustrated in the figure below:



The *pink* graph is what the user is doing physically with the push-button. The *blue* shows how the bit logic values reflect what the user is doing physically (**`bits 2-4`**). The *green* shows how this same information is conveyed for the *edge-triggered* bits (**`bits 5-7`**).

Example:

The following example shows how monitoring the switch state through its *edge-triggered* bit might be useful (*see next page*):

```
// Assume the TINY module is properly opened.

// Assume the LCD module is properly opened.

unsigned int i = 0, j = 0;

unsigned char sensors;

// Infinite LOOP...
while( 1 ) {

  // Delay a little so that the 'Tiny is not overwhelmed with requests.
  TMRSRVC_delay( 100 );  // Wait 100ms.

  // Get state of all sensors.
  sensors = ATTINY_get_sensors();

  if ( sensors & SNSR_SW3_EDGE )

        LCD_printf( "S3 depressed! #%d\n", i++ );

  else if ( sensors & SNSR_SW4_EDGE )

        LCD_printf( "S4 depressed! #%d\n", j++ );

} // end while()
```

The example above shows how monitoring a switch through its *edge-trigger* bit can be used in a loop. When you press a push-button (**S3** in this example), you should see the message printed only ONCE per button-push. If you tried the same code above and used `SNSR_SW3_STATE`, then the message would print over and over again for as long as the button remains depressed. That may or may not be what you want.

Also notice a very important detail – note that we request sensor state via `ATTINY_get_sensors()` *once* per loop iteration, then do bit-wise checking against the variable '`sensors`'. This should always be the preferred method for doing such tests. Note also the delay inserted to prevent the loop from running too fast and overwhelming the 'Tiny support MCU from too many state data queries than it can handle.

The following *macro-constants* can be used to bit-wise testing with the bits returned from `ATTINY_get_sensors()` function:

`SNSR_IR_LEFT`	- Use to refer to Left IR bit position.
`SNSR_IR_RIGHT`	- Use to refer to Right IR bit position.
`SNSR_SW3_STATE`	- Use to refer to S3 (state) bit position.
`SNSR_SW4_STATE`	- Use to refer to S4 (state) bit position.
`SNSR_SW5_STATE`	- Use to refer to S5 (state) bit position.
`SNSR_SW3_EDGE`	- Use to refer to S3 (edge) bit position.
`SNSR_SW4_EDGE`	- Use to refer to S4 (edge) bit position.
`SNSR_SW5_EDGE`	- Use to refer to S5 (edge) bit position.

> **Note:** In case it isn't yet obvious – usage of this function is *highly* discouraged. Users should now make use of `ATTINY_get_IR_state()` and `ATTINY_get_SW_state()` to obtain state data from these peripherals – these new functions are *loop-safe*.

The `ATTINY_set_RC_servos()` Function

Format:

```
void ATTINY_set_RC_servos( unsigned int RCS0_pos,
                           unsigned int RCS1_pos,
                           unsigned int RCS2_pos,
                           unsigned int RCS3_pos,
                           unsigned int RCS4_pos )
```

Description:

This function allows you to *simultaneously* set the position of all 5 servos *if* attached.

Input Arguments:

`RCS0_pos` to `RCS4_pos` – You pass to this argument a value between **400** and **2100** that specifies the rotational position of the servo.

The **400** and **2100** are the *default* extreme limits, which were determined experimentally with the currently supported *standard servos* from PARALLAX. These servos have a rotational span of approximately 180-degrees with values **~400** corresponding to one extreme and values around **~2100** corresponding to the other. However, please keep in mind that your results may vary – perhaps greatly. Determining the physical rotational correspondence to the values you supply to this function can only be determined *experimentally*, such as, for example, to determine what the value would have to be to get your RC servo to sit at its *center* position. You'll just have to write some code to "test" things out.

These limits can be modified via `ATTINY_set_RC_limits()` function.

Example:

```
// Assume the TINY module is properly opened.

// Suppose we have three servos attached on our CEENBoT
// on connectors RC0, RC1 and RC2 -- then the following call
// will position RC0 to one extreme; RC1 to approximately the 'center';
// and RC2 to the other extreme.
ATTINY_RC_servos( 400, 1100, 1900, 0, 0 );
```

The `ATTINY_set_RC_servo()` Function

Format:

```
void ATTINY_set_RC_servo( RCSERVO_ID which, unsigned int RCS_pos )
```

> **Note:** Note this is the *singular* version to `ATTINY_set_RC_servos()` – *plural*.

Description:

This function allows you to control the position of a *specific* servo without affecting the position of the others.

(*Continued on next page*)

(*Continued from previous page*)

Input Arguments:

`which` – Specifies *which* servo is affected. It must be one of the following enumerated constants:

> `RC_SERVO0` – Specify position for RC Servo 0.
> `RC_SERVO1` – Specify position for RC Servo 1.
> `RC_SERVO2` – Specify position for RC Servo 2.
> `RC_SERVO3` – Specify position for RC Servo 3.
> `RC_SERVO4` – Specify position for RC Servo 4.

`RCS_pos` – Specify the *position* for the RC servo.

> **Note:** Please refer to the '*Input Arguments*' section for `ATTINY_set_RC_servos()` function for important information regarding values for this parameter.

Example:

```
// Assume the TINY module has been properly opened.

ATTINY_set_RC_servo( RC_SERVO0, 400  );
ATTINY_set_RC_servo( RC_SERVO1, 1100 );
ATTINY_set_RC_servo( RC_SERVO2, 1900 );
```

The `ATTINY_set_RC_limits()` Function

Format:

```
void ATTINY_set_RC_limits( unsigned int min,
                           unsigned int max,
                           RCSERVO_ID   which )
```

Description:

This function can be used to set the *servo limits* imposed on the positional values that can be supplied via `ATTINY_set_RC_servos()` and `ATTINY_set_RC_servo()` functions. The *real* practical use of this function is actually to further *restricts* the currently impose limits and not necessarily to *expand* them. Although extending the lower limit (from say **`400`** to **`0`**) is certainly doable, it is not practical to extend the upper limit from its default of **`~2100`** since this would force the pulse-width of the signal used to control the servo to exceed its period! The function is provided for convenience, but you should avoid using it if possible.

(*Continued on next page*)

(*Continued from previous page*)

Input Arguments:

`min` – The minimum limit to allow for specifying servo position. (default is **400**).
`max` – The maximum limit to allow for specifying servo position. (default is **2100**).
`which` – Specifies which servo the limits will be applied to. It must be one of the following enumerated constants:

`RC_SERVO0` – Specify position for RC Servo 0.
`RC_SERVO1` – Specify position for RC Servo 1.
`RC_SERVO2` – Specify position for RC Servo 2.
`RC_SERVO3` – Specify position for RC Servo 3.
`RC_SERVO4` – Specify position for RC Servo 4.

Example:

Suppose, through *experimentation*, you determine that to limit your first two servos to a 90-degree rotational span, your values must remain between 500 and 700 (*Note: numbers totally made up!*). Then you would do the following:

```
// Assume the TINY module is properly opened.

// Set limits first.
ATTINY_set_RC_limits( 500, 700, RC_SERVO0 );
ATTINY_set_RC_limits( 500, 700, RC_SERVO1 );

// Okay, now you can use your servos!
```

The `ATTINY_get_SW_state()` Function

Format:

```
BOOL ATTINY_get_SW_state( ATTINY_SW which )
```

Description:

The `ATTINY_get_SW_state()` function can be used to query the state of one of the three on-board switches on the CEENBoT. The function is *loop-safe*, which means it can be inserted in a loop where it can run as fast as possible without slowing the rest of your program down by the necessary insertion of delays that was needed with `ATTINY_get_sensors()`, whose usage is now discouraged. An internal [software] mechanism prevents this function from querying switch state data from the 'Tiny support micro-controller at a rate no greater than 20 times per second, ensuring that the 'Tiny is always able to 'keep up'. This function should be the preferred method for obtaining such state data.

(*Continued on next page*)

(*Continued from previous page*)

Input Arguments:

`which` – Specifies the switch whose state (status) you wish to know. It must be one of the following enumerated constants:

'`ATTINY_SW3`' – To get the current state of Switch 3 (SW3).
'`ATTINY_SW4`' – To get the current state of Switch 4 (SW4).
'`ATTINY_SW5`' – To get the current state of Switch 5 (SW5).

The switch designations **`SW3`**, **`SW4`**, and **`SW5`** correspond to the labeling of the same on the CEENBoT's controller PCB board.

Returns:

Returns a value of type `BOOL`, being '`TRUE`' (non-zero) if the switch is active (depressed), or '`FALSE`' (a value of zero) if the switch is inactive (released).

When a switch is depressed, the function will return '`TRUE`' only once. Any additional queries on the *same* switch will return '`FALSE`' until the user releases the push-botton and depresses it again, at which point it will return '`TRUE`' again, and the cycle repeats. This allows the function to be inserted in loop, but still obtain the expected result or behavior (see 'Example' section below).

Example:

The following example shows how switch state data can be inserted in loop safe-manner, with no delays necessary:

```
unsigned short int i = 0;
unsigned short int j = 0;
unsigned short int k = 0;

while( 1 )
{

    if ( ATTINY_get_SW_state( ATTINY_SW3 ) )
    {

        LCD_printf( "SW3 Pushed: %d\n", ++i );

    } // end if()

    else if ( ATTINY_get_SW_state( ATTINY_SW4 ) )
    {

        LCD_printf( "SW4: Pushed: %d\n", ++j );

    } // end else-if()
```

(*Continued on next page*)

(*Continued from previous page*)

```
        else if ( ATTINY_get_SW_state( ATTINY_S5 ) )
        {

            LCD_printf( "SW5: Pushed: %d\n", ++k );

        } // end else-if()

    } // end while()
```

The `while()` loop in the above example will execute as fast as possible. If you push down on **SW3**, the string **"SW3: Pushed: 1"** will print on the LCD display. If you want to see the message again, you have to release the button and then push it again, at which point it will read **"SW3: Pushed: 2"**, etc. This better coincides user-expected behavior. The above example assumes the LCD subsystem module was previously opened for use.

The ATTINY_get_IR_state() Function

Format:

```
void ATTINY_get_IR_state( ATTINY_IR which )
```

Description:

The ATTINY_get_IR_state() function can be used to query the state of one of the CEENBoT's two infra-red (IR) sensors, and thus determine if there are object obstructions in the path of the CEENBoT. The function is *loop-safe*, which means it can be inserted in a loop where it can run as fast as possible without slowing the rest of your program down by the necessary insertion of delays that was needed with ATTINY_get_sensors(), whose usage is now discouraged. An internal [software] mechanism prevents this function from querying IR state data from the 'Tiny support micro-controller at a rate no grater than 25 times per second, ensuring that the 'Tiny is always able to 'keep up'. This function should be the preferred method for obtaining such state data.

Input Arguments:

which – This argument specifies the IR sensor whose state (status) you wish to know. It must be one of the following enumerated constants:

'ATTINY_IR_LEFT'	– To get the current state of the LEFT IR sensor.
'ATTINY_IR_RIGHT'	– To get the current state of the RIGHT IR sensor.
'ATTINY_IR_EITHER'	– To find out if either the LEFT IR or RIGHT IR (or BOTH) are active.
'ATTINY_IR_BOTH'	– To find out if *both*, the LEFT IR and RIGHT IR are simultaneously active.

Returns:

Function returns a value of type BOOL, being 'TRUE' (non-zero) if the IR requested is active (being blocked by an object), or 'FALSE' (a value of zero) otherwise.

(*Continued on next page*)

(*Continued on previous page*)

Example:

The following comprehensive example shows how to get the CEENBoT to turn right when the right IR sensor is 'tripped'; turn left when the left IR sensor is tripped, and back up a finite distance when both the left and right IR sensors are simultaneously tripped, if ever:

```
#include "capi324v221.h"

void CBOT_main( void )
{

    STEPPER_open();     // Open stepper services.

    // Enter the infinite loop...
    while( 1 )
    {

        // Check if BOTH IR's are 'tripped'... (this rarely happens).
        if ( ATTINY_get_IR_state( ATTINY_IR_BOTH ) )
        {

            // Back up!
            STEPPER_move_stwt( STEPPER_BOTH,
                STEPPER_REV, 400, 200, 400, STEPPER_BRK_OFF,
                STEPPER_REV, 400, 200, 400, STEPPER_BRK_OFF );

        } // end else-if()

        // ... if not, check if just the LEFT IR is 'tripped'...
        else if ( ATTINY_get_IR_state( ATTINY_IR_LEFT ) )
        {

            // Turn Left!
            STEPPER_move_stwt( STEPPER_BOTH,
                STEPPER_REV, 150, 250, 400, STEPPER_BRK_OFF,
                STEPPER_FWD, 150, 250, 400, STEPPER_BRK_OFF );

        } // end else-if()

        // ... if not, then finally check if the RIGHT IR is 'tripped'.
        else if ( ATTINY_get_IR_state( ATTINY_IR_RIGHT ) )
        {

            // Turn Right!
            STEPPER_move_stwt( STEPPER_BOTH,
                STEPPER_FWD, 150, 250, 400, STEPPER_BRK_OFF,
                STEPPER_REV, 150, 250, 400, STEPPER_BRK_OFF );

        } // end else-if()

    } // end while()

} // end CBOT_main()
```

Chapter 13: The TMRSRVC (Timer Service) Subsystem Module

The **TMRSRVC** subsystem module provides timing services that go a step above your usual *delay* functions. In particular the timer service makes *controlled program execution* based on timed events easily.

Module at a Glance

Description

The **TMRSRVC** module, or *timer service* module provides *millisecond* accurate services and features. It provides your typical *delay* function except unlike typical *delay* mechanisms, which rely on repeated *loops* of some sort, the delays are achieved in hardware using the microcontroller's **Timer0** device embedded in the MCU. In addition, the timer service module also provides some basic, but quite-useful operating system-like timing features that allow program execution to be controlled based on time designations, thereby saving precious processor cycles.

Another similar feature is the ability to declare what are called *timer events*. These are functions that are executed in *interrupt-context* which get automatically called when a timer criteria triggers it – however, it should be noted that use of interrupt-context timer events is heavily discouraged.

The timer service module is an the most integral feature of the CEENBoT-API and it is started by default.

Modular Dependencies

The **TMRSRVC** module is *opened by default.* It has no other dependencies.

Hardware Dependencies

The following MCU-specific dependencies exist:

- **Timer0** – The timer service relies on this embedded MCU device – it is reserved for use by the CEENBoT-API given and it's an integral part of its operation. The user should not attempt to access this device by bypassing the API.
- NO *interrupt service routines* (ISRs) related to **Timer0** may be used nor invoked for the same reasons, whether through the API (by way of CBOT_ISR()), or outside of it by bypassing (e.g., by using the ISR()) macro).

Function List Summary

- TMRSRVC_open() - Open the *timer service* module.
- ~~TMRSRVC_close()~~ - Close the *timer service* module.
- TMRSRVC_delay() - Standard *millisecond* delay function.
- TMRSRVC_new() - Create, submit and start a *timer object*.
- TMRSRVC_stop_timer() - Kills a current timer object.
- TMRSRVC_wait_on_stop() - Causes execution to *hold* until the timer object is destroyed.

Macro-Function Summary for the new CRC (Call Rate Controlled) Features supported by the TMRSRVC Subsystem Module as of API Revision v1.08.000R

- DECLARE_CRC_FUNCTION() – Declare a CRC function and its corresponding timer object.
- DEFINE_CRC_FUNCTION() – Define a CRC function (function *body*).
- START_CRC_FUNCTION() – Start the timer object of a corresponding CRC function.
- STOP_CRC_FUNCTION() – Stop the timer object of a corresponding CRC function.
- ON_CRC_TIME() – Trigger a CRC function when its corresponding timer is up.

Function Reference

The `TMRSRVC_open()` Function

Format:

```
SUBSYS_OPENSTAT TMRSRVC_open( void )
```

Description:

Function acquires and initializes resources needed for operation of the timer service (**TMRSRVC**). You *must* call this function first with a successful 'open' before invoking any other function provided by this module.

Returns:

Returns a structure of type SUBSYS_OPENSTAT whose field entries indicate the status of the open request and the subsystem that resulted in an error (when, and *if* an error occurs). See *Chapter 1, Procedure for Opening Modules Before Use* for examples on how to handle the *open status*.

> **Note:** Please note that the timer service (**TMRSRVC**) is open by default.

The `TMRSRVC_close()` Function

Format:

```
void TMRSRVC_close( void )
```

Description:

You most certainly *cannot* close this service. The function *does* exist – it's just empty and it doesn't do anything at the present.

> **Note:** The *timer service* CANNOT be stopped. It is an integral part of the CEENBoT-API.

The `TMRSRVC_delay()` Function

Format:

```
TMRNEW_RESULT TMRSRVC_delay( TIMER16 delay_ms )
```

Description:

This is your standard *delay* function. You can use this delay to delay a specified number of *milliseconds*. The `TIMER16` type defaults to a `signed int` (16-bits on the AVR-GCC platform). Consequently the maximum delay value is **`32767`** milliseconds (or **`37.767`** seconds). It's primary function is for *millisecond-scale* delay requests. Note the function will BLOCK (that is, it will *hold*) until the delay interval has been fulfilled – just like your typical delay does.

Input Arguments:

`delay_ms` – The length of the delay that should elapse before program execution resumes. The value must be between **`0ms`** and **`32767ms`**.

You can also use the helper macro-function `TMR_SECS()` to specify the value to this parameter in units of *seconds* instead of *milliseconds* (see example below).

Returns:

Function returns a value of type `TMRNEW_RESULT`. This value need not be checked. It simply indicates the result of the *internally created timer object* used to count time. However, if you are curious, the function will return one of the following enumerated constants:

`TMRNEW_OK`	– No errors in creating and submitting the *timer object*.
`TMRNEW_ERROR`	– Some unknown, non-categorized error.
`TMRNEW_OUTOFMEMORY`	– The internal timer object could not be created because the system was not able to dynamically allocate memory for it.

Example:

```
// Assume the LED subsystem is properly open.

// Assume the LCD subsystem is properly open.

// Recall the TMRSRVC is open by default.

LCD_clear();

// Print a message.
LCD_printf( "Hello World!\n" );

// Wait 2 seconds...
TMRSRVC_delay( TMR_SECS( 2 ) );

// Print another message.
LCD_printf( "Starting LED toggle.\n" );

// Wait another two seconds.
TMRSRVC_delay( TMR_SECS( 2 ) );
```

(*Continued from previous page*)

```
// Clear the screen and print a new message.
LCD_clear();
LCD_printf( "Toggling LED!\n" );

// Toggle the GREEN led every 250ms... forever...
while( 1 ) {

  // Toggle the LED.
  LED_toggle( LED_Green );

  // Wait 250ms...
  TMRSRVC_delay( 250 );

} // end while()
```

It should be noted that the above example certainly achieves the task of toggling the LED once every **250ms**. This may be sufficient for simple cases. The problem with this approach however is that although the LED does toggle once every **250ms**, the entire program is paused for **250ms** as TMRSRVC_delay() executes. This may or may not be desirable.

For example, there may be a need to toggle the LED every **250ms**, but there may be other parts of the program that should execute as fast as possible, and not be 'held back' by your placement of the TMRSRVC_delay(250) in your program. If you find yourself in such a scenario, then you should consider some of the more advance constructs afforded by the **TMRSRVC** module. For that, take a look at the information for the function that follows, TMRSRVC_new() and in particular, read though the examples thoroughly (discussed *next*).

Finally, note the usage of the TMR_SECS() macro, which converts the value from units of *seconds* to *milliseconds*, so that:

TMRSRVC_delay(**TMR_SECS(2)**) is equivalent to TMRSRVC_delay(**2000**)

Also, the following equivalent helper macros are available for use which perform what their namesake suggests:

- **TMRSRVC_delay_secs()** – For delaying in units of *seconds*. (e.g., TMRSRVC_delay_secs(3)).
- **TMRSRVC_delay_ms()** – For delaying in units of *milliseconds* (e.g., TMRSRVC_delay_ms(200)).

Just keep in mind these are all available for your use. Their only advantage is that these 'helper' macros are *self-documenting* – meaning, that it is 'obvious' what the units of the delays are merely by looking at the function name, whereas something like TMRSRVC_delay(200) may not be immediately clear.

The `TMRSRVC_new()` Function

Format:

```
TMRNEW_RESULT TMRSRVC_new(   TIMEROBJ      *pTimerObject,
                             TMR_FLGS      validNotifyFlags,
                             TMR_TCMODE    tcMode,
                             TIMER16       nTicks )
```

Description:

This function allows you to start a new *timer* by submitting a *timer object* to the *timer service*. A *timer object* is a structure of type `TIMEROBJ` which has internal fields that keep time and other state data. This object has to be declared by *you*. It is not created dynamically in order to save time and the scarce memory that exists in the MCU.

When you call this function you provide the *address of* your timer object. Obviously, the reason you started the timer in the first place is to perform some action after the timer expires. To do this successfully, you specify this function an argument called *timer notify flags*, which specify in what way you'd like to be notified of a timer *expiration* event, of which there are two ways:

- The timer service will set an internal *timer expiration flag* in your *timer object* to let you know the timer has expired. You can then use the function `TMRSRVC_on_TC()` to execute a function at this time in a *process context*.
- The timer service will automatically call a *timer event function*, which is a function you declare and define via the `TMR_EVENT()` macro that you register and associate with your *timer object* via `TMRSRVC_REGISTER_EVENT()` macro which gets executed in *interrupt context*.

After specifying the way in which timer expiration notification takes place, you then specify the *terminal count* mode. This specifies what happens when the timer expires, of which two options are possible:

- Kill the *timer* from the *timer service* once it expires (in other words, *run once*).
- Re-register the *timer* with the same settings and run it again (in other words, *restart it*).

The final parameter allows you to specify the time interval in *milliseconds*.

After `TMRSRVC_new()` succeeds, it inserts a *reference* to your *timer object* with current settings in the internal list of timer objects. As soon as this happens, which happens almost instantly (function doesn't BLOCK), program execution continues on its merry way whereby at some later time, your timer will *expire* and some action will probably take place.

The 'Example' section gives some examples on how you might use the timer services.

(*Continued on next page*)

(*Continued from previous page*)

Input Arguments:

`pTimerObject` – You pass to this argument the *address of* a structure of type `TIMEROBJ`, which you must have declared somewhere in your program.

`validNotifyFlags` – This argument allows you to specify the manner in which your timer object will be notified (or the action that will take place) upon timer *expiration*. It must be one of the following:

`TMRFLG_NOTIFY_FLAG` – Tell the *timer service* to set a internal flag when the timer expires.

`TMRFLG_NOTIFY_FUNC` – Tell the *timer service* to trigger a *timer event* when the timer expires. If you specify this option you *must* declare and define a *timer event* function using the `TMR_EVENT()` macro and then register (or associate it) with your timer object by using `TMRSRVC_REGISTER_EVENT()` macro (see the 'Example' sections to see how this is accomplished).

Note that you can also specify *both* parameters by *bit-wise OR-ing* them together:

```
TMRFLG_NOTIFY_FLAG | TMRFLG_NOTIFY_FUNC
```

which says you want the internal flag to be set in addition to triggering the interrupt-context timer event.

`tcMode` – This argument specifies the *terminal count* mode. That is – what should happen when the timer expires. You must specify one of the following:

`TMRTCM_RUNONCE` – As the name implies – run the timer *once* (a *one-shot* timer) and kill it – that is, remove it from the *timer service* when it expires.

`TMRTC_RESTART` – Restart the timer once it expires. This re-registers the timer with the timer service with the original settings present in the *timer object*.

`nTicks` – This parameter specifies the number of *ticks* that should elapse before the timer expires and notifies the internal flag or triggers an event. Presently, each *tick* is worth 1ms, so you can specify anywhere between **`0ms`** up to **`32767ms`** (or **`32.767 seconds`**). You can use the `TMR_SECS()` macro to specify the time in units of *seconds* instead of *milliseconds*.

Returns:

Function returns a value of type `TMRNEW_RESULT`. This value need not be checked. It simply indicates the result of the *internally created timer object* used to count time. However, if you are curious, the function will return one of the following enumerated constants:

`TMRNEW_OK` – No errors in creating and submitting the *timer object*.
`TMRNEW_ERROR` – Some unknown, non-categorized error.
`TMRNEW_OUTOFMEMORY` – The internal timer object could not be created because the system was not able to dynamically allocate memory for it.

(*Continued on next page*)

(*Continued from previous page*)

Example:

Let us take a look at some 'code snippets' to show the various ways in which the timer service can be used.

The first example shows a *better* way to toggle an LED every **250ms** *just* as we tried to do in the example shown for `TMRSRVC_delay()` function previously discussed – check it:

```
// Recall the TMRSRVC is open by default - you do NOT need to call 'TMRSRVC_open()'.

TIMEROBJ  led_timer; // Our timer object for the LED.

// Start the new timer.  We want this timer to be triggered every
// 250ms forever -- we will toggle the LED ONLY when it's TIME!
TMRSRVC_new( &led_timer, TMRFLG_NOTIFY_FLAG, TMRTCM_RESTART, 250 );

// AT this point the timer is already running...

// We just wait for the internal flag called 'tc' to be set as follows:
while( 1 ) {

  // IF it's time....
  if ( led_timer.tc ) {

         // Toggle the GREEN LED.
         LED_toggle( LED_Green );

         // Reset the notification flag for the next 'round' - THIS IS IMPORTANT!!!
         led_timer.tc = 0;

  } // end if()

} // end while()
```

Now let me explain you why this is a *better* example than the example shown for `TMRSRVC_delay()` where we try to accomplish the same thing – *toggle* the LED. In the example for `TMRSRVC_delay()` we run in a loop and *delay* program execution before toggling the LED. The problem with *that* example is that `TMRSRVC_delay()` BLOCKS while the delay is taking effect – this is wasteful, because we can't do anything else while we wait for the delay to complete or expire. In contrast, in the example given *here*, program execution does NOT BLOCK. This is one example of how the timer service can be used for *controlled program execution*. We check if it's time and only then, do we execute any code – in our case, toggle the LED. We do this quickly, and we exit – we don't *block*. We then reset the internal flag (called '`tc`') back to zero. If you forget to do that, you'll never be notified again!

The next example expands a bit on the same above – but this time we'll do it more *elegantly* by using the `TMRSRVC_on_TC()` macro-function. In addition, we'll now toggle the RED LED every **500ms**. Moreover, we'll put the work of toggling the LEDs inside of our functions `foo()` and `bar()`. The examples show LED toggling, but you can extend this idea to executing your *own* code every **250ms** and **500ms** correspondingly. Let's take a look:

```
#include "capi324v221.h"

// -------------- prototypes:
void foo( void );
void bar( void );

// -------------- functions:
void foo( void ) {

  LED_toggle( LED_Red );

}

void bar( void ) {

  LED_toggle( LED_Green );

}

// -------------- CBOT_main:
void CBOT_main( void ) {

  TIMEROBJ     red_led_timer; // Will be triggered every 500ms once started.
  TIMEROBJ     grn_led_timer; // Will be triggered every 250ms once started.

  // Start the corresponding timers.
  TMRSRVC_new( &red_led_timer, TMRFLG_NOTIFY_FLAG, TMRTCM_RESTART, 500 );
  TMRSRVC_new( &grn_led_timer, TMRFLG_NOTIFY_FLAG, TMRTCM_RESTART, 250 );

  // Enter the while loop and ONLY execute 'foo()' every 500ms,
  // and 'bar()' every 250ms.
  while( 1 ) {

      TMRSRVC_on_TC( red_led_timer, foo() ); // If 'TIME'... execute 'foo()'.
      TMRSRVC_on_TC( grn_led_timer, bar() ); // If 'TIME'... execute 'bar()'.

         // ... do other stuff here ...

  }

} // end CBOT_main()
```

The example above shows how you should *always* approach controlled program execution using timer objects. This should always be the *preferred* way, instead of using *timer events*, which get executed in *interrupt context* (more on this later). In any case, note that the `while()` loop executes forever, and as fast as the MCU allows it to. However, `foo()` and `bar()` DON'T get triggered every time on each loop iteration. Instead, `foo()` is triggered *only* every **500ms** and `bar()` is triggered every **250ms**. While the the timers are counting – you are free to do "other stuff" because functions are not blocking – which would be a drawback if you chose to do the same thing with *delays* instead. This is a *huge* advantage.

The `TMRSRVC_on_TC()` macro-function automatically checks to see if the '`tc`' field of the specified timer object is set, and if so, it triggers the expression in the second argument. After the expression is executed, '`tc`' will be automatically reset to zero, so you don't have to. Compare this in contrast with the first example that was previously given.

Note that you can use `TMRSRVC_on_TC()` macro-function even if the function being triggered either takes, or returns a parameter (or both). For example, suppose `foo()` takes two arguments, `a` and `b`, and returns this into a variable called `c`. You could have done the following:

```
TMRSRVC_on_TC( red_led_timer, c = foo( a, b ));
```

So as you can see, this is a versatile feature.

The next example shows how to declare and define *timer events*. Timer events are functions that get triggered when your timer expires. The timer event function also has high-priority because it gets executed in *interrupt-context*. This benefit comes with tremendous amount of danger. The timer service uses **`Timer0`** – also, and the *stepper subsystem* also uses **`Timer0`**. If you execute a timer event function in an interrupt context and your function takes too long to execute, you're keeping both the timer service and the stepper engine from running! My advice to you is to DO EVERYTHING YOU CAN TO AVOID USING THIS FEATURE. However, IF, you feel that using timer events is exactly the thing you're looking for you better make sure it is *short* and *sweet*. “Short-and-sweet” means specifically that your timer event function should take less than **`1ms`** to execute and complete. If you can't guarantee this, then you'll mess everything else and the API will become unresponsive. You've been warned!

Now that I've given you ample warning, I'll show you how to *toggle* the green LED once again, but this time, using *timer events*. So, here it goes:

```
#include "capi324v221.h"

// -------------- prototypes:
TMR_EVENT( led_toggle_evt );

// -------------- functions:
TMR_EVENT( led_toggle_evt ) {

  LED_toggle( LED_Green );

}


// -------------- CBOT_main:
void CBOT_main( void ) {

  TIMEROBJ     toggle_timer;

  // Attach our 'timer event' with our 'timer object'.
  TMRSRVC_REGISTER_EVENT( toggle_timer, led_toggle_evt );

  // Start 'toggle_timer' to trigger every 250ms.
  TMRSRVC_new( &toggle_timer, TMRFLG_NOTIFY_FUNC, TMRTCM_RESTART, 250 );

  while( 1 ); // Never leave.

} // end CBOT_main()
```

(*Continued on next page*)

In this example we declare and define *timer event* functions by using the `TMR_EVENT()` macro. These are declared as if they were functions, outside of `CBOT_main()`. Then as per usual, we declare our timer object and then *attach* our timer event to this timer object by way of `TMRSRVC_REGISTER_EVENT()` macro. The first argument specifies the timer object I'm referring to and the second the timer event function to attach to it.

Finally, I start this timer, this time specifying that I want the *event function* to be notified (triggered), which I do by specifying `TMRFLG_NOTIFY_FUNC` as an option.

After completion of `TMRSRVC_new()`, you should see the LEDs toggle indefinitely.

Notice that the *while* loop is empty. This is because our function is being triggered in the background, in *interrupt context*. Here the timer event function's body is "short-and-sweet" and I'm certain I can meet the **`1ms`** deadline since it doesn't take much to toggle an LED.

As elegant as *timer event* functions might be, it is best to execute your timed functions in a process-context instead of interrupt-context by using timer objects along with timer flag notification and the `TMRSRVC_on_TC()` macro-function. This is the recommended *best*, *safe* way to do it.

A Word on Declaring Timer Objects

You need to be *very* conscious of *where* you declare your timer objects. Most of the examples that implement timer objects shown thus far are declared in *local scope*. However, you'll also notice that an infinite *while* loop is in place so that the function in which the timer object is declared doesn't terminate. For example:

```
void CBOT_main( void )
{

  TIMEROBJ myTimer;   // Declare here.

  // ... use 'myTimer' somehow...

  // Keep CBOT_main() from ever terminating.
  while( 1 );

}
```

Suppose in the last example shown, where I showed you how to use *timer events* we did NOT have a *while* loop at the very end. What do you suppose will happen? The timer will start, the timer events will trigger, then `CBOT_main()` finishes and exits. At this point, our timer object which was called `toggle_timer` would go out scope! The program will most likely crash and burn.

The moral of the story is this – if you declare a timer object in *local scope*, you need to make sure you finish using the timer before the function exists – if the timer is started so that it always restarts itself, at some point you'll have to kill the timer via `TMRSRVC_stop_timer()`. The SAFEST way, of course, is to declare your timer objects as *global* outside of any one function. That way you can guarantee your timer objects will never go out of scope and your programs won't crash.

The `TMRSRVC_stop_timer()` Function

Format:

```
void TMRSRVC_stop_timer( TIMEROBJ *pWhich )
```

Description:

Use this function to change the *terminal count* mode of a timer object from `TMRTCM_RESTART` to `TMRTCM_RUNONCE`. Then, on the *next* terminal count down or timer expiration, the specified timer object will be removed from the *timer service*. This essentially allows you to stop a timer that was previously started to run perpetually. If the timer object was previously created in *local scope*, it is highly recommended that you wait for the timer to be completely destroyed by using the `TMRSRVC_wait_on_stop()` macro (see 'Example' section below).

Input Arguments:

`pWhich` – Pass to this argument the *address of* a timer object of type `TIMEROBJ` that you have already started via `TMRSRVC_new()`, but that you now wish to stop.

Example:

This example, which still surrounds the toggling of an LED shows how to do it 10 times – that is, we use a timer object to toggle an LED, but only do so 10 times, at which point, we terminate it (see *next* page).

```
#include "capi324v221.h"

// -------------- globals:
TIMEROBJ toggle_timer;

// -------------- prototypes:
void foo( void );

// -------------- functions:
void foo( void ) {

  static unsigned int toggle_count = 0;

  // Toggle the LED.
  LED_toggle( LED_Green );

  // Increment the toggle count.
  toggle_count++;

  // Check if we have done this 10 times.
  if ( toggle_count == 10 )
  {

         // If so... stop the timer here.
         TMRSRVC_stop_timer( &toggle_timer );

         // Wait here and make sure the timer is destroyed before proceeding.
         TMRSRVC_wait_on_stop( toggle_timer );

   } // end if()

} // end foo()

// -------------- CBOT_main:
void CBOT_main( void ) {

  // Start 'toggle_timer' to trigger every 250ms.
  TMRSRVC_new( &toggle_timer, TMRFLG_NOTIFY_FUNC, TMRTCM_RESTART, 250 );

  while( 1 ) {

         // If 'TIME'... then trigger 'foo()'.
         TMRSRVC_on_TC( toggle_timer, foo());

  }

} // end CBOT_main()
```

Note, how in this example, we were forced to declare our timer object using *global scope*. This is the only way we can start our timer in `CBOT_main()`, but be able to stop it in `foo()` when the time was right. Also take note of how we *wait* and make sure the timer is fully destroyed before proceeding. Normally, this is something you want to do if timer object was created in *local scope* to prevent the function from going out of scope before the timer service subsystem has had a chance to destroy the timer object and remove it from the internal timer list. Take a look at the info on '`TMRSRVC_wait_on_stop()`' (*next* page) for an additional example.

The `TMRSRVC_wait_on_stop()` Macro-Function

Format:

```
void TMRSRVC_wait_on_stop( TIMEROBJ timer_obj )
```

Description:

This function causes program execution to *hold* (or *wait*) until a timer object pending deletion has been fully destroyed. This is useful in cases where the timer object was previously created in *local scope* (i.e., inside of a function's body instead of *globally* using *global scope*). It allows the timer subsystem module to properly destroy the timer object before it goes out of scope. It only makes sense to call this after '`TMRSRVC_stop_timer()`' has been invoked on the same timer object. Otherwise, the function will BLOCK forever waiting for the destruction of timer object that will *never* happen.

Input Arguments:

`timer_obj` – Pass to this argument the name (NOT the address) of the timer object to wait on.

Example:

Suppose we have a need for a timer object and we only need it temporarily inside of our 'fake' function called '`do_timer_stuff()`'. Here's how we create, and [safely] destroy the timer:

```
// Suppose we're inside of some function:
void do_timer_stuff( void )
{

    TIMROBJ my_timer;

    // Start the timer.
    TMRSRVC_new( &my_timer, TMRFLG_NOTIFY_FLAG, TMR_TCM_RESTART, 150 );

    // ...Do some stuff here with the timer....

    // Okay, we're done, so stop the timer.
    TMRSRVC_stop_timer( &my_timer );

    // Let's be safe and wait HERE until the timer is destroyed.
    TMRSRVC_wait_on_stop( my_timer );

}
```

Introduction to CRC (Call-Rate Controlled Functions)

The TMRSRVC Subsystem Module now supports a new feature called *CRC Functions*. CRC stands for *call-rate controlled* and it allows one to determine how 'quickly' a user's function is invoked (i.e., *called).* There's really nothing new to using CRC functions other than the given benefit that you don't have to deal with timer objects directly – that stuff is now *hidden* with CRC functions. Everything that you can do with CRC functions can be readily applied with all the functions previously discussed (See the examples for '`TMRSRVC_new()`'). CRC functions bring up a 'notch' the *elegance* factor in achieving the same task.

The CRC features are actually *macros* wrapped around some of the existing `TMRSRVC_xxxxx()` functions. Before discussing each macro-function in detail, let's simply redo one of the examples given for 'TMRSRVC_new()' whereby we create two functions to toggle the RED LED and the GREEN LED at different rates (**`500ms`** and **`250ms`** respectively). The example accomplishes EXACTLY the same thing that was done in that example, but it merely shows how the problem is solved by declaring and implementing CRC functions:

```
#include "capi324v221.h"

// ---------------------------------- Prototypes:

// Same as 'void toggle_red( void );'
DECLARE_CRC_FUNCTION( toggle_red,   void );

// Ssame as 'void toggle_green( void );'
DECLARE_CRC_FUNCTION( toggle_green, void );

// ---------------------------------- Functions:

DEFINE_CRC_FUNCTION( toggle_red )
{

    // Just toggle the RED led.
    LED_toggle( LED_Red );

}

DEFINE_CRC_FUNCTION( toggle_green )
{

    // Just toggle the GREEN led.
    LED_toggle( LED_Green );

}
```

(*Continued on next page*)

(*Continued from previous page*)

```
// ---------------------------------- Main:
void CBOT_main( void )
{

    LED_open();     // Open the LED subsystem module.

    START_CRC_FUNCTION( toggle_red,   500 );
    START_CRC_FUNCTION( toggle_green, 250 );

    // 'while()' loop runs hundreds of thousands of times per second.  However...
    while( 1 )
    {

        ON_CRC_TIME( toggle_red );      // Only runs once every 500ms.
        ON_CRC_TIME( toggle_green );     // Only runs once every 250ms.
    }

}
```

Lets walk through this example.

We start at the prototypes declarations. Here, we use the 'DECLARE_CRC_FUNCTION()' macro to declare our two functions toggle_red() and toggle_green(), both of which take a 'void'. The first argument to DECLARE_CRC_FUNCTION() is always the name of your CRC function. All parameters thereafter refer to the *arguments* of your function. More about this later.

Also, 'DECLARE_CRC_FUNCTION()' must always occur in *global* scope. Just like you can't declare prototypes inside of functions, you cannot declare a CRC function inside of another one.

Next, after we declare our function prototypes, we then *define* the body of each of the CRC functions, which we do via the DEFINE_CRC_FUNCTION() macro. Inside of the function's *body* you put whatever it is that you're going to do, which must be call-rate controlled. In our case, we have the one-liner that toggles the GREEN or RED LED correspondingly.

Finally, we move on to CBOT_main(), where the action occurs. Before our CRC functions can be invoked, we must 'activate' them, which we do via START_CRC_FUNCTION() macro. It is here where we determine the rate at which the functions are invoked by specifying a *delay* between calls. So toggle_red() will be invoked once every **500ms** (or *twice* a second), while toggle_green() will be invoked every **250ms** (or *four* times per second).

Once active, the functions must be invoked inside of the ON_CRC_TIME() macro. For example, ON_CRC_TIME(toggle_red) basically states that if the specified **500ms** have already elapsed, to invoke toggle_red() function, otherwise, skip it and go to the next line, etc.

CRC functions represent the most elegant use of the timer services for timed process-context execution.

Let us look at another example. This example shows how parameters can be passed to our CRC functions for more advanced usage. The example toggles the GREEN LED 10 times and then stops toggling. It keeps track of how many more times to toggle by application of *pointers*:

```
#include "capi324v221.h"

// ---------------------------------- Prototypes:

// Same as 'void toggle_green( unsigned short int *pTimes, BOOL *pExit );'
DECLARE_CRC_FUNCTION( toggle_green, unsigned short int *pTimes, BOOL *pExit );

// ---------------------------------- Functions:

// Same as 'void toggle_green( unsigned short int *pTimes, BOOL *pExit );'
DEFINE_CRC_FUNCTION( toggle_green, unsigned short int *pTimes, BOOL *pExit )
{

    // Just toggle the GREEN led.
    LED_toggle( LED_Green );

    // Decrement the number of toggles left.
    --( *pTimes );

    // If zero, note that we should exit the loop.
    if ( *pTimes == 0 )

        *pExit = TRUE;

}

// ---------------------------------- Main:
void CBOT_main( void )
{

    unsigned short int toggle_count = 10;

    BOOL exit_loop = FALSE;

    LED_open();     // Open the LED subsystem module.

    START_CRC_FUNCTION( toggle_green, 250 );

    while( 1 )
    {

        ON_CRC_TIME( toggle_green, &toggle_count, &exit_loop );

        if ( exit_loop == TRUE )

            break;
    }

    STOP_CRC_FUNCTION( toggle_green );

}
```

The `DECLARE_CRC_FUNCTION()` Macro

Format:

```
DECLARE_CRC_FUNCTION( function_name, ... )
```

Description:

Function transparently performs the following two things for the user: a) it declares a *timer object* called *function_name*_timer, and b) also declares the prototype for the function given by *function_name*. Because of this, the user must make sure that 'DECLARE_CRC_FUNCTION()' is declared in *global scope* – meaning, you can't declare this inside of any function.

Input Arguments:

function_name – The first parameter represents the name that will be given to the timer object and the name of the CRC function being declared.

... – Any parameters that follow after the first constitute the variadic *parameters* of the function given by *function_name*. If your CRC function has no parameters, then you should at least specify `void` after the function name (see the 'Example') section below.

Example:

Example 1: Suppose you want to declare a CRC function with the following signature:

```
void foo( unsigned short int a, signed short int b, char * c );
```

You would declare your CRC function as follows:

```
// Equivalent to 'void foo( unsigned short int a, signed short int b, char * c );'
DECLARE_CRC_FUNCTION( foo, unsigned short int a, signed short int b, char * c );
```

Example 2: Suppose you want to declare a CRC function with the following signature:

```
void bar( void );
```

You would declare your CRC function as follows:

```
// Equivalent to 'void bar( void );'.
DECLARE_CRC_FUNCTION( bar, void );
```

Etc.

The `DEFINE_CRC_FUNCTION()` Macro

Format:

```
DEFINE_CRC_FUNCTION( function_name, ... )
```

Description:

This macro is used to defined the *body* of your CRC function previously declared via `DECLARE_CRC_FUNCTION()` macro. The function name and the arguments that follow must be exactly identical to the arguments given to `DECLARE_CRC_FUNCTION()`.

Input Arguments:

`function_name` – The first parameter represents the name of the CRC function. It *must* be the same as the function name given to the corresponding `DECLARE_CRC_FUNCTION()` macro.

`...` – Any parameters that follow after the first constitute the variadic *parameters* of the function given by *function_name*. If your CRC function has no parameters, then you should at least specify `void` after the function name (see the 'Example') section below.

Example:

```
// Same as 'void add( char a, char b, char *result )
DECLARE_CRC_FUNCTION( add, char a, char b, char *result );

DEFINE_CRC_FUNCTION( add, char a, char b, char *result )
{

    // Add up 'a' and 'b', and return in 'result'.
    *result = a + b;

}
```

The `START_CRC_FUNCTION()` Macro

Format:

```
START_CRC_FUNCTION( function_name, call_rate_interval_ms )
```

Description:

This macro is used to 'activate' the CRC function's corresponding timer object for the specified CRC function given by *`function_name`* . It also assigns the CRC function a *call rate interval* in *milliseconds* which essentially determines how often the function is allowed to be invoked in a 'loop' scenario.

(*Continued on next page*)

Input Arguments:

`function_name` – The first parameter represents the name of the CRC function. It *must* be the same as the function name given to the corresponding `DECLARE_CRC_FUNCTION()` macro.

call_rate_interval_ms – You must specify for this parameter the interval *in milliseconds*, which must elapse before the function is allowed to be invoked again. The maximum rate at which the CRC function is invoked will be the reciprocal of this value. For example, if you specify a call rate interval of **`250ms`**, then the reciprocal is **`1/250ms = 4`**, which means the CRC function will not be allowed to be called at a rate faster than *four* times per second.

Example:

Following example shows how a CRC function is used to count up the number of seconds elapsed. Note this is not necessarily a 'time-accurate' method for keeping track of time – it's just an example.

```
#include “capi324v221.h”

// Declare CRC function.
DECLARE_CRC_FUNCTION( count, void );

// Define CRC function.
DEFINE_CRC_FUNCTION( count, void )
{

    static unsigned short int seconds = 0;

    LCD_printf_RC( 3, 0, "Seconds: %d\t", seconds++ );

}

// Usage.
void CBOT_main( void )
{

    LCD_open();

    // 'count()' will be allowed to run only once every second (1000ms).
    START_CRC_FUNCTION( count, 1000 );

    while( 1 ) {

        ON_CRC_TIME( count );

    }

}
```

The `STOP_CRC_FUNCTION()` Macro

Format:

```
STOP_CRC_FUNCTION( function_name )
```

Description:

This macro can be used to *deactivate* a CRC function. It essentially stops the associated timer object and waits for its destruction. Note that the function *will* block until destruction of the underlying timer object has completed. This means, for example, if you initially specified a CRC function with a 1-second interval, then it will take up to one second for it to be destroyed.

Note that under most circumstances it is NOT necessary to invoke this macro, unless you wish to start the CRC function again (via `START_CRC_FUNCTION()`) with a different call rate interval value.

Input Arguments:

function_name – The first parameter represents the name of the CRC function. It *must* be the same as the function name given to the corresponding `DECLARE_CRC_FUNCTION()` macro.

Example:

The example on page 12-17 shows how the `STOP_CRC_FUNCTION()` can be applied.

The `ON_CRC_TIME()` Macro

Format:

```
ON_CRC_TIME( function_name )
```

Description:

The `ON_CRC_TIME()` macro is where the 'magic' happens. This macro performs a few things. First it checks the timer object associated with the named CRC function to see if the time interval specified with `START_CRC_FUNCTION()` has expired. If so, it then invokes *function_name* and once it completes, it clears the timer object's internal flag bit so that the process can repeat again on the next loop iteration.

Perhaps the most important aspect of this macro is that it is traditionally implemented in some kind of 'loop' construct, such as a `while()`, `do-while()`, or `for()` loop. It doesn't make sense NOT to use it in a loop.

Input Arguments:

function_name – The first parameter represents the name of the CRC function. It *must* be the same as the function name given to the corresponding `DECLARE_CRC_FUNCTION()` macro.

Example:

The example given for `START_CRC_FUNCTION()` is repeated on the next page.

```
#include "capi324v221.h"

// Declare CRC function.
DECLARE_CRC_FUNCTION( count, void );

// Define CRC function.
DEFINE_CRC_FUNCTION( count, void )
{

    static unsigned short int seconds = 0;

    LCD_printf_RC( 3, 0, "Seconds: %d\t", seconds++ );

}

// Usage.
void CBOT_main( void )
{

    LCD_open();

    // 'count()' will be allowed to run only once every second (1000ms).
    START_CRC_FUNCTION( count, 1000 );

    while( 1 ) {

        // Invoke 'count()' only once per second.
        ON_CRC_TIME( count );

    }

}
```

Chapter 14: The UART Subsystem Module

> The **UART** Subsystem Module gives the user the ability to use the on-board *universal asynchronous receiver transmitter* devices embedded on the MCU.

Module at a Glance

Description

The current MCU contains two USARTS (*universal synchronous receiver transmitter*) devices embedded as MCU peripherals. This module exposes a set of functions which give the user the ability to use these devices in a simplified manner. However, when using the API functions, these serial communication devices can only be used in *asynchronous* mode – hence the reason we refer to this module as the UART and *not* USART. This stems from the fact that the *clock* pin that would allow synchronous operation of the USART is not exposed in the CEENBoT hardware, making synchronous usage of these embedded peripherals not possible on the CEENBoT.

> **Note:** The **UART** subsystem module allows the serial device to be used in *asynchronous* mode only.

Modular Dependencies

The UART module *must* be manually opened by the user. It has no other modular dependencies.

Hardware Dependencies

- I/O Port pin **PD0** on **PORTD** (for **RXD0**)
- I/O Port pin **PD1** on **PORTD** (for **TXD0**)
- I/O Port pin **PD2** on **PORTD** (for **RXD1**)
- I/O Port pin **PD3** on **PORTD** (for **TXD1**)

Function List Summary

- **UART_open()** - Opens the UART subsystem module for use.
- **UART_close()** - Closes the UART subsystem module when no longer needed.
- **UART_set_TX_state()** - Used to enable or disable the *transmitter* portion of the UART device.
- **UART_set_RX_state()** - Used to enable or disable the *receiver* portion of the UART device.
- **UART_configure()** - Used to configure the UART before use (e.g., parity, character size, baud).
- **UART_set_timeout()** - Used to control how long 'UART_receive()' should wait before giving up.
- **UART_transmit()** - Used to transmit data.
- **UART_receive()** - Used to receive data.
- **UART_has_data()** - Used to determined whether the UART has data awaiting to be read.
- **UART_printf()** - Used to send 'formatted text' via the UART serial device.
- **UART_printf_PGM()** - Used to send 'formatted text' via the UART serial device. String data, however, is stored in *Program Memory* instead of SRAM.

'Helper' Macros Summary

In addition to the set of functions provided by the **UART** subsystem module, these additional *helper macros* are provided which perform *exactly* the same as their function counterparts, except that you don't have to specify *which* UART (always the first parameter) device is being affected. For example, `UART_printf()` requires that you specify *which* UART is being affected (almost all functions do):

```
UART_printf( UART_UART0, “Hello World\n” );
```

With the *helper macro*, however, the above shortens a bit:

```
UART0_printf( “Hello World\n” );
```

It is less 'verbose' in my opinion – in fact, I *highly* encourage that you use *these* instead! Almost all original functions have 'helper' macro counterparts, though not all.

- **`UART0_printf()`** – Same as `UART_printf( UART_UART0, ... )`.
- **`UART0_transmit()`** – Same as `UART_transmit( UART_UART0, ... )`.
- **`UART0_receive()`** – Same as `UART_receive( UART_UART0, ... )`.
- **`UART0_has_data()`** – Same as `UART_has_data( UART_UART0, ... )`.
- **`UART0_printf_PGM()`** – Same as `UART_printf_PGM( UART_UART0, ... )`.

- **`UART1_printf()`** – Same as `UART_printf( UART_UART1, ... )`.
- **`UART1_transmit()`** – Same as `UART_transmit( UART_UART1, ... )`.
- **`UART1_receive()`** – Same as `UART_receive( UART_UART1, ... )`.
- **`UART1_has_data()`** – Same as `UART_has_data( UART_UART1, ... )`.
- **`UART1_printf_PGM()`** – Same as `UART_printf_PGM( UART_UART1, ... )`.

> **Note:** The new *'helper'* macros were written much after the UART documentation was conceived. For this reason, many of the examples in this chapter do NOT show usage of these 'helper' macros. However, feel free to replace the original function calls with their 'helper' macro equivalents. This practice is highly encouraged!
>
> In some of the examples in this chapter, I've tried to point out areas where you can make use of the 'helper' macros. These examples are highlighted in *blue*.

Function Reference

The `UART_open()` Function

Format:

```
SUBSYS_OPENSTAT UART_open( UART_ID which )
```

Description:

Function acquires and initializes resources needed for operation of the **UART** subsystem module. You *must* call this function first with a successful 'open' before invoking any other function provided by this module.

> **Note:** Note that unlike all `<MODULE>_open()` family of functions discussed thus far, *this* particular version actually requires a parameter to be provided as an input argument.

Input Arguments:

`which` – You must specify which **UART** device to open using one of the following enumerated constants:

`UART_UART0` – Open **UART0** device for use.
`UART_UART1` – Open **UART1** device for use.

Returns:

Returns a structure of type `SUBSYS_OPENSTAT` whose field entries indicate the status of the open request and the subsystem that resulted in an error (when, and *if* an error occurs). See *Chapter 1, Procedure for Opening Modules Before Use* for examples on how to handle the *open status*.

Example:

```
#include "capi324v221.h"

void CBOT_main( void )
{

  SUBSYS_OPENSTAT opstat;

  // Open the UART module to use UART0 device.
  opstat = UART_open( UART_UART0 );

  if ( opstat.state == SUBSYS_OPEN )
  {

         // ... DO UART STUFF ...

  } // end if()

} // end CBOT_main()
```

Note in the example above that you have to specify *which* **UART** device to open when you invoke the corresponding 'open' function. So far, this is the only exception where you'll see an 'open' function where an input argument is required.

The `UART_close()` Function

Format:

```
void UART_close( UART_ID which )
```

Description:

Function deallocates and releases resources being used by the **UART** subsystem module. No other functions should be invoked once the subsystem module is closed. Note that you must specify *which* **UART** device you're attempting to close.

Input Arguments:

`which` – You must specify which **UART** device to close using one of the following enumerated constants:

`UART_UART0` – Close **UART0** device.
`UART_UART1` – Close **UART1** device.

The `UART_set_TX_state()` Function

Format:

```
void UART_set_TX_state( UART_ID which, UART_STATE uart_state )
```

Description:

This function can be used to enable or disable the *transmitter* portion of the specified UART device. When the specified transmitted is *enabled,* the corresponding I/O pins are taken over by the UART device.

Input Arguments:

`which` – You must specify which **UART** device is affected by using one of the following enumerated constants:

`UART_UART0` – Specify **UART0** device.
`UART_UART1` – Specify **UART1** device.

`uart_state` – Must be one of the following enumerated constants:

`UART_DISABLE` – The specified UART transmitter will be disabled.
`UART_ENABLE` – The specified UART transmitter will be enabled.

Example:

See the example for `UART_configure()`.

The `UART_set_RX_state()` Function

Format:

```
void UART_set_RX_state( UART_ID which, UART_STATE uart_state )
```

Description:

This function can be used to enable or disable the *receiver* portion of the specified UART device. When the specified receiver is *enabled*, the corresponding I/O pins are taken over by the UART device.

Input Arguments:

`which` – You must specify which **UART** device is affected by using one of the following enumerated constants:

- `UART_UART0` – Specify **UART0** device.
- `UART_UART1` – Specify **UART1** device.

`uart_state` – Must be one of the following enumerated constants:

- `UART_DISABLE` – The specified UART receiver will be disabled.
- `UART_ENABLE` – The specified UART receiver will be enabled.

Example:

See the 'Example' section for `UART_configure()`.

The `UART_configure()` Function

Format:

```
void UART_configure( UART_ID     which,
                     UART_DBITS  data_bits,
                     UART_SBITS  stop_bits,
                     UART_PARITY parity,
                     UART_BAUD   baud_rate )
```

Description:

This function allows the user to *set-up* and *configure* the specified UART device BEFORE use. Preferably, it should follow after having issued `UART_open()`. Function allows you to set the *character size*, *stop bits*, *parity mode* and *baud rate* of the specified UART device. Note that configuration parameters apply to both, transmission and reception of data.

Input Arguments:

`which` – You must specify which **UART** device is affected by using one of the following enumerated constants:

- `UART_UART0` – Specify **UART0** device.
- `UART_UART1` – Specify **UART1** device.

(*Continued on next page*)

(*Continued from previous page*)

`data_bits` – Specifies the *character size*. You must specify one of the following enumerated constants:

`UART_5DBITS` – Character size is 5-bits.
`UART_6DBITS` – Character size is 6-bits.
`UART_7DBITS` – Character size is 7-bits.
`UART_8DBITS` – Character size is 8-bits.

`stop_bits` – Specifies the number of *stop bits* to use. You must specify one of the following enumerated constants:

`UART_1SBIT` – Use 1 stop bit.
`UART_2SBITS` – Use 2 stop bits.

`parity` – Specifies the *parity* format. You must specify one of the following enumerated constants:

`UART_NO_PARITY` – NO parity used.
`UART_EVEN_PARITY` – Use *even* parity.
`UART_ODD_PARITY` – Use *odd* parity.

`baud_rate` – Specify the *baud rate* in *bits-per-second* (bps). You can specify any value for this parameter up to the maximum allowable theoretical baud rate which is 1.2Mbps (or **`1200000`**). However, it has been tested up to **`0.5Mbps`** (Mega-bits/sec). For reference, typical standard baud rates are shown below:

- **9600**
- **14400**
- **19200**
- **28800**
- **38400**
- **56000**
- **57600**
- **112500**

Example:

The following example shows how to open, configure the UART and enable the *transmitter* portion.

```
#include "capi324v221.h"

void CBOT_main( void )
{

    // Open the UART module to use UART0 device.
    // We'll assume it just 'opens' and will disregard the returned
    // status code.
    UART_open( UART_UART0 );
```

(*Continued on next page*)

(*Continued from previous page*)

```
        // Configure the UART device -- we want:
        //  8-bits,
        //  1-stop bit,
        //  NO parity,
        //  9600bps baud.
        //
        UART_configure( UART_UART0,
                     UART_8DBITS, UART_1SBIT, UART_NO_PARITY, 9600 );

        // Enable the transmitter -- we just want to transmit:
        UART_set_TX_state( UART_UART0, UART_ENABLE );


        //  At this point we're ready to go and start transmitting data
        //  via 'UART_transmit()' function...


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

    } // end CBOT_main()
```

The `UART_set_timeout()` Function

Format:

```
void UART_set_timeout( UART_ID which, UART_TIMEOUT timeout_sec )
```

Description & Motivation:

When you invoke `UART_receive()` to acquire data via the UART's receiver, the function will wait UNTIL data arrives if there isn't data already awaiting to be read from the internal buffer. If no data ever arrives for whatever reason and no timeout mechanism existed, your code would completely lock-up because `UART_receive()` would sit there indefinitely waiting for data that will never arrive. Luckly, `UART_receive()` has a *timeout mechanism* – which means that `UART_receive()` will wait for data to arrive (if it hasn't yet arrived) until either data DOES arrive, or until a timeout occurs – at which point, `UART_receive()` will give up and return with nothing for you. *This* function, then – `UART_set_timeout()` – allows you to set what this *timeout* period is going to be in units of *seconds*. By default, the timeout period is automatically set to **5** seconds. However, if you needed to change that, then *this* function will allow you to do so.

Input Arguments:

`which` – You must specify which **UART** device is affected by using one of the following enumerated constants:

> `UART_UART0` – Specify **UART0** device.
> `UART_UART1` – Specify **UART1** device.

`timeout_sec` – The number of seconds to wait before a *timeout* occurs. Must be between **0** and **30** seconds.

The `UART_transmit()` Function

Format:

```
UART_COMM_RESULT UART_transmit( UART_ID which, unsigned char data )
```

Description:

Use this function to transmit data out of the specified UART device. Make sure the specified UART device is already opened, properly configured, and *enabled*.

Input Arguments:

`which` – You must specify which **UART** device is affected by using one of the following enumerated constants:

- `UART_UART0` – Specify **UART0** device.
- `UART_UART1` – Specify **UART1** device.

`data` – The data to transmit (8-bits max). The number of relevant bits depends on the *character size* specified via `UART_configure()` – recall character sizes are 5, 6, 7, or 8 bits.

Returns:

Function returns a value of type `UART_COMM_RESULT`, which can consist of one of the following enumerated constants:

- `UART_COMM_OK` – Communication went OK.
- `UART_COMM_ERROR` – Some unknown error has occurred.
- `UART_COMM_TX_FULL` – Transmitter is not ready to accept new data.
- `UART_COMM_TIMEOUT` – Communication timeout while waiting for transmitter to be ready.

Note that inspecting the return value is not as important as when you are *receiving*.

Example:

```
// Assume the UART0 device has been properly opened,
// configured, and enabled.

unsigned int i;

const unsigned char data_to_send[] = { 0x04, 0x08, 0x15, 0x16, 0x23, 0x42 };

// Transmit these 'NUMBERS' or we'll be LOST.
for ( i = 0, i < 6, i++ )

    // Send it.
    // Note: you can also say: 'UART0_transmit( data_to_send[ i ] )'
    UART_transmit( UART_UART0, data_to_send[ i ] );
```

The `UART_receive()` Function

Format:

`UART_COMM_RESULT UART_receive( UART_ID which, unsigned char *pDest )`

Description:

Use this function to receive data from the specified UART device. Make sure the specified UART device is already opened, properly configured, and *enabled*.

Input Arguments:

`which` – You must specify which **UART** device is affected by using one of the following enumerated constants:

- `UART_UART0` – Specify **UART0** device.
- `UART_UART1` – Specify **UART1** device.

`pDest` – You must pass to this argument the *address of* a variable of type `unsigned char`, where the incoming data will be stored once received.

Returns:

Function returns a value of type `UART_COMM_RESULT`, which can consist of one of the following enumerated constants:

- `UART_COMM_OK` – Communication went OK.
- `UART_COMM_ERROR` – Some unknown error has occurred.
- `UART_COMM_TIMEOUT` – Communication timeout while waiting for receiver data.

Example:

```
// Assume the UART0 device has been properly opened,
// configured, and enabled.

unsigned int i;

unsigned char temp;              // Store a single byte.
unsigned char buffer[ 10 ];      // Store multiple bytes.

// Read 10 bytes...
for ( i = 0, i < 10, i++ ) {

    // Get the data and store it on 'temp'.
    // Note: You can also say 'UART0_receive( &temp )'.
    UART_receive( UART_UART0, &temp );

    // Store it in the buffer.
    buffer[ i ] = temp;

} // end for()
```

The `UART_has_data()` Function

Format:

`BOOL UART_has_data( UART_ID which )`

Description & Motivation:

If you use `UART_receive()`, there's a chance that the function will hang until the *timeout* period has expired *if* data never arrives. To avoid 'waiting', it is possible to simply *check* to see if data has indeed arrive before you make an attempt to actually read it from the *receiver* buffer of the UART. You can do this check with *this* function. `UART_has_data()` will return TRUE, if data has been *indeed* received at which point, it is safe to issue a `UART_receive()` to actually retrieve this data with no waiting 'penalty' because you know for sure there is 'some' data to read.

Input Arguments:

`which` – You must specify which **UART** device is affected by using one of the following enumerated constants:

> `UART_UART0` – Specify **UART0** device.
> `UART_UART1` – Specify **UART1** device.

Returns:

Function returns a value of type `BOOL`. It returns `TRUE` if there is data awaiting to be read via `UART_receive()`, or `FALSE` if there is no data because it hasn't been received, or because it is still arriving.

Example:

```
// Assume the UART0 device has been properly opened,
// configured, and enabled.

unsigned int bytes_received = 0;    // Bytes in.

unsigned char temp;              // Store a single byte.
unsigned char buffer[ 10 ];      // Store multiple bytes.

// Keep waiting until all 10 bytes have been received.
do {

    // First check IF there is data to read...
    // Note: You can say 'if ( UART0_has_data() ) == TRUE )' also.
    if ( UART_has_data( UART_UART0 ) == TRUE )
    {

        UART_receive( UART_UART0, &temp );      // Read it...

        buffer[ bytes_received ] = temp;      // Store it...

        bytes_received++                      // Increment byte count.

    } // end if()

} while( i < 10 );
```

The `UART_printf()` Function

Format:

```
void UART_printf( UART_ID which, const char *str_fmt, ... )
```

Description:

This function is meant to be used when you connect the CEENBoT's serial port to a PC's serial terminal. It's primary purpose is for debugging, using the PC's *terminal* as an alternate output display instead of using the LCD display. It works just like the *printf* function, within the limitations of the embedded system.

> **Note:** It is not within the scope of this document to teach you how to use `printf()`.

Input Arguments:

`which` – You must specify which **UART** device is affected by using one of the following enumerated constants:

`UART_UART0` – Specify **UART0** device.
`UART_UART1` – Specify **UART1** device.

`str_fmt` – A constant string which can contain format specifiers and escape sequences.

`...` – Additional *variadic* parameters traditional to the '`printf()`' style.

Example:

```
// Assume the UART0 device has been properly opened,
// configured, and enabled.

unsigned char data;

UART_printf( "Waiting for data...\n" );

while( 1 ) {

    if ( UART_has_data( UART_UART0 ) == TRUE )
    {

        UART_receive( UART_UART0, &data );

        // Note: You can say 'UART0_printf( “Received = 0x%X\n” )'.
        UART_printf( UART_UART0, "Received = 0x%X\n" );

        break;  // Break out of the while loop.

    } // end if()

} // end while()

// Note: You can also say 'UART0_printf( “Data received – test DONE.\n” )'.
UART_printf( UART_UART0, "Data received -- test DONE.\n" );
```

This is a *simple* example. Note that you can do with `UART_printf()` pretty much anything that you can do with the standard `printf()` facility. Note, however that not all features may be present or supported given that you're working in an embedded environment and NOT on a PC.

The `UART_printf_PGM()` Function

Format:

```
void UART_printf_PGM( UART_ID which, const char *str_fmt, ... )
```

Description:

This function works *exactly* like `UART_printf()`, with the only difference that the *constant string* is stored in *Program Memory* instead of SRAM. This is advantageous because strings can very quickly eat up SRAM. Since Program Memory is much larger than SRAM, it should be the preferred way of 'printing strings' via the UART. You need to declare your constant strings to reside in Program Memory via the `PSTR()` macro (See the 'Example' section below).

Input Arguments:

`which` – You must specify which **UART** device is affected by using one of the following enumerated constants:

`UART_UART0` – Specify **UART0** device.
`UART_UART1` – Specify **UART1** device.

`str_fmt` – A constant string which can contain format specifiers and escape sequences.

`...` – Additional *variadic* parameters traditional to the '`printf()`' style.

Example:

```
// Assume UART0 has been properly opened and configured.
// Somewhere in 'CBOT_main()'...

unsigned int value = 20;

// Print some string.
UART_printf_PGM( UART_UART0, PSTR( “Hello World\n” ) );

// Print the contents of a variable.
UART_printf_PGM( UART_UART0, PSTR( “Value = %d\n” ), value );
```

You can also take advantage of the 'helper' macros:

```
// Print some string.
UART0_printf_PGM( PSTR( “Hello World\n” ) );

// Print the contents of a variable.
UART0_printf_PGM( PSTR( “Value = %d\n” ), value );
```

In either case, note usage of the `PSTR()` macro! This is important! You **CANNOT** use any of the `UART_xxxxx_PGM()` functions without declaring your constant strings with `PSTR()`.

Chapter 15: The USONIC (Ultrasonic) Subsystem Module

> The **USONIC** subsystem modules exposes functional services that facilitate the use of the PARALLAX *Ping* Ultrasonic Device specifically, when attached to the CEENBoT.

Module at a Glance

Description

The **USONIC** subsystem module exposes functional services that facilitate the user of an Ultrasonic device when (and *if*) attached to the CEENBoT. The module is written to specifically support the *Ping* Ultrasonic module from *PARALLAX, Inc*. For details on using this device, please refer to the following *data-sheet* from PARALLAX:

- **PING)))™ Ultrasonic Distance Sensor (#28015) – *PARALLAX, Inc***

You can search for this document *on-line*.

It is not within the scope of this document to discuss how the Ultrasonic sensor works, but essentially, there is a single digital I/O pin on the Ultrasonic sensor (in addition to power and ground). A pulse of a specified duration is sent to this sensor to initiate and trigger an Ultrasonic wave. The Ultrasonic sensor (automatically) switches from transmitter to receptor after it emits an ultrasonic wave that travels through air. At the same time the same I/O pin you used to trigger the Ultrasonic sensor switches from *input* to *output* (on the Ultrasonic sensor) and a logic signal becomes active to let the host MCU (or any controlling device) that the unit has fired an ultrasonic wave. As soon as this waves bounces from some surface or object, comes back and is *received* by the ultrasonic sensor, the pulse on the I/O pin goes back to being inactive to let you know it has received the 'echo'. As you can see, this requires the designated I/O pin on the controlling device or MCU to start as an *output pin* and then immediately be switched to an *input pin* after the PING Ultrasonic Sensor has been *triggered*.

Moreover, the user must measure the time that that this I/O pulse was active, and the length (or width) of this pulse is linearly correlated to the *time* the ultrasonic wave traveled, bounced, and came back.

To measure the *pulse-width* accurately, the **USONIC** module makes use of the **SWATCH** (the *stopwatch*) service (see respective chapter for that), since the stopwatch service was precisely written for this reason – measuring the pulse-width of the 'echo-pulse' emitted by the PING Ultrasonic Sensor.

Modular Dependencies

The **USONIC** module *must* be manually opened. It has, also, the following *modular* dependencies:

- **SWATCH** – The **USONIC** depends on *accurate* time measurements from **SWATCH**.

You must ensure these dependencies are available by properly opening them before using any functions exposed by the **USONIC** module.

Hardware Dependencies

- I/O port pin **PA3** on **PORTA** (available via **J3 – Pin 1** on Header Connector)

> **Note!!!:** Prior to API revision **v1.08.000R** the 'trigger' pin was assigned to **PA4**. However this has now been changed (moved) to **PA3** to accommodate interfacing with the TI-calculator interface which makes use of port pins **PA[7..4]**. User's with API programs targeting earlier versions of the API will have to move their 'trigger' wire from **PA4** (old way) to **PA3** (new way).

(*Continued on next page*)

(*Continued from previous page*)

Presently this is the *assigned* port pin for triggering the Ultrasonic Sensor.

> **Note:** Presently, this pin assignment is *fixed* – you cannot use a different I/O pin to achieve the same task of triggering the Ultrasonic Sensor.

Function List Summary

- `USONIC_open()` - Open and initialize the USONIC subsystem module.
- `USONIC_close()` - Close and release the USONIC subsystem module.
- `USONIC_ping()` - Triggers PARALLAX Ultrasonic Sensor, the 'wave' travel-time is measured.

Function Reference

The `USONIC_open()` Function

Format:

`SUBSYS_OPENSTAT USONIC_open( void )`

Description:

Function acquires and initializes resources needed for operation of the **USONIC** module. You *must* call this function with a successful 'open' before invoking any other functions provided by this module.

Returns:

Returns a structure of type `SUBSYS_OPENSTAT` whose field entries indicate the status of the open request and the subsystem that resulted in an error (when, and *if* an error occurs). See *Chapter 1, Procedure for Opening Modules Before Use* for examples on how to handle the *open status*.

Example:

```
#include "capi324v221.h"

void CBOT_main( void )
{

    SUBSYS_OPENSTAT ops_swatch;
    SUBSYS_OPENSTAT ops_usonic;

    // Open the STOPWATCH and the USONIC modules.
    ops_swatch = STOPWATCH_open();   // NOTE: USONIC depends on SWATCH.
    ops_usonic = USONIC_open();

    // Only continue if they were successfully opened.
    if ( ( ops_swatch.state == SUBSYS_OPEN ) &&
         ( ops_usonic.state == SUBSYS_OPEN ) )
    {

        // ... Do USONIC Stuff...

    } // end if()

} // end CBOT_main()
```

The `USONIC_close()` Function

Format:

`void USONIC_close( void )`

Description:

Function deallocates and releases resources being used by the **USONIC** subsystem module. No other functions should be invoked once the subsystem module is closed.

The `USONIC_ping()` Function

Format:

```
SWTIME USONIC_ping( void )
```

Description:

Function triggers the attached Ultrasonic Sensor to emit a 'ping' or ultrasonic wave. The function will wait until either the echo of this wave is received at some point, or a timeout occurs.

Returns:

Function returns a value of type `SWTIME` (*stopwatch time*), which defaults to an `unsigned short int`. This value represents the number of *ticks* corresponding for the time for a complete ROUND TRIP for the ultrasonic sound wave – that is *transmission*, *reflection*, and *reception*. The time duration of each 'tick' depends on the granularity of the stopwatch service. At the time this documentation was being written, each tick is worth **`10us/tick`**. So multiplying the resulting `SWTIME` value by **`10`**, will give you the *round-trip* time in *microseconds*.

Make sure that when you multiply the SWTIME value by 10, that you store the result in a type that is larger than `unsigned short int`. In the AVR platform, this is usually the `unsigned long int` (as 'plain' `int` defaults to 16-bits also in the AVR platform). See the 'Example' section below.

To measure the distance-to-target you use the following generalized formula:

$$d \;=\; \frac{1}{2} \cdot v_s t$$

where d is the distance, v_s the speed of sound in *air* in the appropriate units and t the time in *seconds*. However, recall that the time returned by `USONIC_ping()` after conversion is in units of *microseconds*, and NOT *seconds*. So you must adjust your formula accordingly to accept these units. We multiply by 1/2 because we want the time from source to target, not the entire *round-trip* time.

For example, suppose we wish to compute the distance-to-target in units of *cm* (*centimeters*). The speed of sound in air is approximately 344.8m/s, or 34480cm/s. So we have the following:

$$d \;=\; \frac{1}{2}(34480) \cdot t \;=\; 17240 \cdot t$$

where t is still given in *seconds*. To specify in *microseconds* we multiply the above by **1×10^{-6}**, and thus, we have the following:

$$d_{\text{cm}} \;=\; 0.01724 \cdot t_{\mu\text{s}}$$

Now, you can plug in $t_{\mu\text{s}}$, the value you get after you multiply the returned `SWTIME` by **`10`**, to convert *ticks* to *microseconds*. A similar approach can be used to obtain distance in other units. But that is not the scope of this document – this is a *function reference* manual, and not “Introduction to Physics” ;o)

Example:

The following is a *comprehensive* example on computing the distance-to-target using `USONIC_ping()` in *centimeters*.

```
// Assume the SWATCH and USONIC modules have been properly opened.

// Assume the LCD module is properly opened.

unsigned long int usonic_time_us;
SWTIME            usonic_time_ticks;
float             distance_cm;

// Ping once.
usonic_time_ticks = USONIC_ping();

// Convert to 'us'.
usonic_time_us = 10 * (( unsigned long int )( usonic_time_ticks ));

// Convert to 'cm'.
distance_cm = 0.01724 * usonic_time_us;

// Print result.
LCD_printf( "Dist = %f\n", distance_cm );
```

> **Note:** The above example uses the `printf()` facility using *floating point* numbers. Using of floating point numbers with the `printf()` facility is NOT enabled by default! Please refer to the *Getting Started* manual, where this topic is addressed!

Note that in the above example, where we convert from *ticks* to *microseconds* by multiplying by **`10`**, we have to *type-cast* from the smaller type to the larger one. This is necessary to let the compiler to treat the smaller storage type as a larger one while it does the conversion and to let the compiler know this *is* our intention.

Chapter 16: The I2C Subsystem Module

This chapter discusses the functional services offered by the I2C/TWI subsystem module provided by the CEENBoT-API.

Module at a Glance

Description

The **I2C** subsystem module provides functional services needed to make use of the MCU's internal *Inter-IC* (I2C) or as ATMEL calls it, *Two-Wire Interface* (TWI) Peripheral Subsystem. The I2C bus is a serial bus comprising of just two signal lines, one to carry the clock signal, and the other to carry data. Because the I2C specification protocol 'embeds' the address of each device in the protocol itself, there is traditionally – and under *certain* circumstances – no need for *chip-select* lines because the addressing scheme is part of the protocol itself. Because of this it is *theoretically* possible to connect up to 127 I2C devices on these two bus lines alone without any additional hardware requirements – provided these devices have been assigned unique I2C addresses by the manufacturer of the I2C device in question – making I2C a simple, elegant solution for serial interfacing tasks. In any case, the I2C subsystem module functional services provided by the CEENBoT-API easily provides the means to access these facilities.

> **Note:** It is not within the scope of this document to discuss the I2C bus, its protocol, nor how it works. It is assumed that users understand the hardware principles behind it. The I2C protocol is a universal standard – it is not specific to the CEENBoT. Users that don't know much of, or haven't even heard of, I2C are suggested to read the following Wikipedia article for insight: http://en.wikipedia.org/wiki/I2C.
>
> This should be followed by consulting the datasheet for the CEENBoT's MCU (presently the **ATmega324**) to learn how I2C (they call it TWI) specifically works on this device.

> **Note:** The terms **I2C** and **TWI** are interchangeable as far as this documentation is concerned.

Modular Dependencies

The **I2C** subsystem module *must* be manually opened by the user as it is *closed* by default. It has no other modular dependencies.

Hardware Dependencies

The I2C peripheral device on the MCU has the following hardware dependencies:

- I/O port pin **PC0** on **PORTC** (for **SCL** line – the *serial clock*)
- I/O port pin **PC1** on **PORTC** (for **SDA** line – the *serial data*)

These two pins comprise the “I2C bus”. The CEENBoT's *controller board* provides multiple 'tap' points to these two pins, available through the following connectors:

On header connector **J3**:

- Pin **12** (I2C – **SCL**)
- Pin **13** (I2C – **SDA**)

(*Continued on next page*)

(*Continued from previous page*)

On both *forward-facing* **DB9** Connectors (**J5** and **J6**):

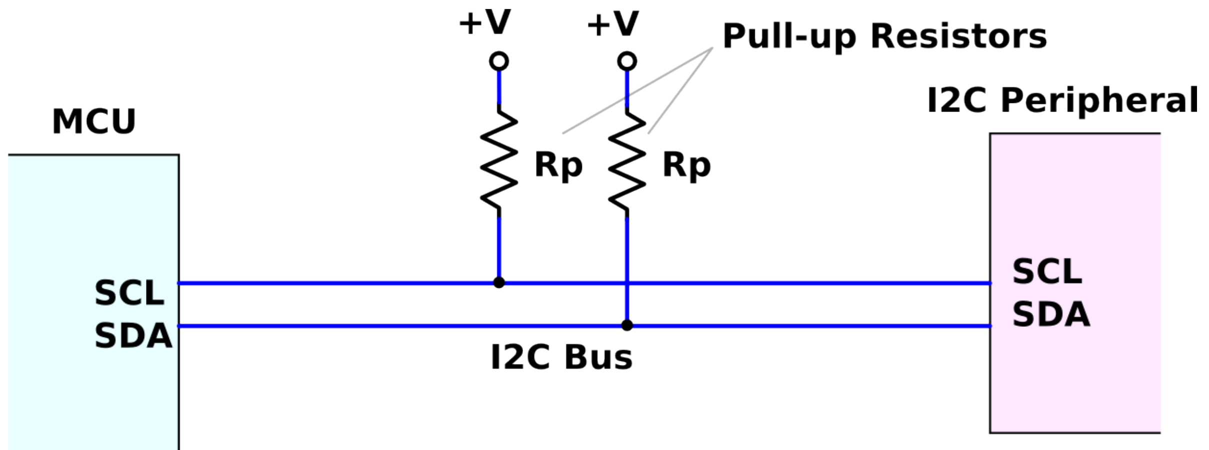
- Pin 8 (I2C – **SCL**)
- Pin 7 (I2C – **SDA**)

> **Note:** The DB9 pins are also shared with the SPI device lines.

Regardless of which pins you choose to 'tap' into, keep in mind these two pins are from the *same* I2C bus – they're not three separate I2C busses. Just one, with three different 'tap points'.

Other Hardware Requirements

Another hardware requirement for the I2C specification is that the bus lines must be pulled up by a resistor as shown in the following diagram:



Again, this is part of the I2C specification, and not a "CEENBoT thing". The resistance value to use for the pull-ups is traditionally around ~1.5K ohms. However, before you immediately make the decision to use 1.5K ohms, just because I said so, you *really* should read the following introductory and insightful article regarding resistor pull-ups and the I2C bus: http://www.dsscircuits.com/articles/effects-of-varying-i2c-pull-up-resistors.html.

For simple circuit designs, it [may] be possible to omit the pull-up resistors, and instead use the *internal pull-ups* provided by the MCU by invoking the API-provided functions `I2C_pullup_enable()`/`I2C_pullup_disable()` respectively, but sometimes the internal resistors are not sufficient, and thus you may be forced to apply your own external pull-up resistors.

> **Note:** It is *highly* recommended that you always provide/use *external* resistors instead. I have found that using the *internal* MCU resistors is a bit 'hit-and-miss'.

Modes of Operation

The I2C protocol allows I2C devices to operate in one of two possible modes – these are: *master* device, or *slave* device. The 'master' device is the device that is in control of the bus, and therefore is the device that both initiates a *communication transaction* with a slave device on the same bus, and the *direction* in which data is to travel (from master-to-slave [writing/sending], from slave-to-master [reading/getting/receiving]).

Consequently, whether you're *writing/sending/transmitting* or *reading/getting/receiving* depends from which device's point of view you're talking about. When the *master* device is transmitting, the *slave* device is receiving and vice versa. Thus, it's important that you are always making this distinction from the get-go, particularly when talking about I2C.

Moreover, it is possible for an I2C device to enable both modes *master* and *slave*. That is, it is able to act as a master of the bus when master, or behave as a slave, when the bus is under the control of another master device on the same bus. What cannot happen, is for both modes to be 'active' at the same time – only *enabled*.

Configuration Settings Before Use

By default, upon opening the I2C subsystem module via `I2C_open()`, the clock rate of the I2C clock signal is setup to run at **`100Khz`**, as this is the traditional clock rate of a lot of I2C devices. However, you may need to alter this setting to a different one. To do that, you have to configure the I2C's *bit rate generator* BRG via the `I2C_set_BRG()` function. Please refer to information in this chapter about the `I2C_set_BRG()` function to understand how to set up the bit rate generator to achieve a different clock rate other than the default.

Once the I2C subsystem is opened – and optionally, the bit rate generator is set to something other than the default – you can immediately begin a *communication transaction* with another I2C device as either “master” or “slave”. Communication transactions as “master” are more straight forward than as “slave” – these are discussed next.

Operating as “Master”

This is the *most* likely scenario that you're going to deal with – operation as 'master' device – which is also the most straight forward method. In this mode, the MCU of the CEENBoT operates as the 'master' I2C device with one or more slave I2C devices, like sensors and such, connected to the same bus. In this mode, you traditionally send a command to a slave I2C device to tell it to do something, or configure the device in some way, and then retrieve data by reading from this same device.

The I2C protocol requires communication transactions from master-to/from-slave to proceed in a certain fashion. When initiating a communication transaction with a slave device, you can operate as either “*Master Transmitter*” (MT) or *“Master Receiver*” (MR). Meaning – in both cases, you're the 'master', and you decide the direction in which data will flow between master and slave devices.

As an example, consider the following superstitious scenario. Let us suppose that you want to access a slave I2C device that responds to address 0x78 – pretend this is a *color sensor*. Suppose further that after you begin communicating with this device, the device expects you to transfer the address of one of the 10 internal registers starting at address 0x20 to 0x29, whose contents will be returned back to the master. Suppose that the register at 0x20 of the slave device contains the *red* color component of the sensor; 0x21 the *green* color component; and 0x22 the *blue* color component. Thus, the goal is to access the color sensor at 0x78, and read its registers at 0x20, 0x21, 0x22 back-to-back because the manufacturer says I can do so in its datasheet (note that not all devices allow this kind of back-to-back access).

The following simplistic code snippet suggests one way you may go about it:

(*Continued from previous page*)

```
// Open the I2C Subsystem Module.
unsigned char red_component   = 0;
unsigned char green_component = 0;
unsigned char blue_component  = 0;

// Open the I2C subsystem module.
I2C_open();

// Initiate a communication transaction w/ the color sensor
// at I2C address 0x78 on the I2C bus as 'MASTER TRANSMITTER'.
if( I2C_MSTR_start( 0x78, I2C_MODE_MT ) == I2C_STAT_OK )
{

    // If we get to this point, this means the I2C slave
    // device responded to our communication request.

    // Tell it we want to start reading its register contents
    // starting at 0x20.
    I2C_MSTR_send( 0x20 );

} // end if()

// We're not done yet -- now we want to READ from the slave
// device, and have it start sending data from 0x20, 0x21, 0x22..., etc.
// So to read, we're going start a new communication transaction, but
// now as 'MASTER RECEIVER'.
if ( I2C_MSTR_start( 0x78, I2C_MODE_MR ) == I2C_STAT_OK )
{

    // Read the red component & acknowledge back.
    I2C_MSTR_get( &red_component, TRUE );

    // Read the green component & acknowledge back.
    I2C_MSTR_get( &green_component, TRUE );

    // Read the blue component, and -DO NOT- acknowledge
    // back, because this is the last byte we're going to read.
    I2C_MSTR_get( &blue_component, FALSE );

    // END the communication transaction with the slave device and
    // relinquish the I2C bus.
    I2C_MSTR_stop();

} // end if()

// At this point, the variables 'red_component', 'green_component',
// and 'blue_component' should have valid data.
```

This is the 'gist' of what it takes to operate as “Master”. You start by 'taking hold of the I2C bus' by initiating a communication transaction with a slave device, whose address you give to '`I2C_MSTR_start()`' function. It is at this point when you dictate whether you're acting as “Master Transmitter” or “Master Receiver”.

At any point in the middle of a communication transaction – that is, while you still have hold of the I2C bus – you can switch the directionality of the communication between being master transmitter and master receiver. Note that transmission makes use of the `I2C_MSTR_send()` function.

When receiving, note that you receive data via the `I2C_MSTR_get()` function. What is interesting about the `I2C_MSTR_get()` function is that you must acknowledge the reception of data except for the last byte. This is the way that the I2C specification dictates it, and is the reason that in the previous code sample, all except the last byte are acknowledged.

The transaction then completes officially when you *relinquish* the I2C bus – which you do by stopping the communication transaction with the `I2C_MSTR_stop()` function.

Operating as "Slave"

Operating as an I2C slave device is a bit trickier than operating in master mode. This is because a slave device has *no* control of the bus. The job of a slave device is to 'sit there' and wait for the master to issue a 'send' or 'get' request to which the slave is obliged to fulfill. Therefore, and consequently, all operations in slave mode occur in interrupt context, inside of the TWI vector ISR (interrupt service routine). The following code snippet shows the coding model you should follow when writing 'slave' I2C code – notice, in particular, that the processing occurs inside of an interrupt service routine ISR for the TWI (recall TWI = I2C) interrupt vector (the second part of the code snippet below). The code written for an I2C slave device is divided in two parts: a) the *initialization* code, and b) the *isr-resident* code:

The initialization code proceeds in the same fashion as in master mode. You open it, and perhaps also configure the baud-rate generator values (if the clock rate is other than **`100KHz`**):

```
void CBOT_main( void )
{

    // open the I2C subsystem.
    I2C_open();

    // Enable device to operate "as slave" and to respond to address 0x50.
    I2C_SLVE_enable( 0x50 );

    // Keep the MCU running.
    while( 1 );

} // end main()
```

Next is the ISR-resident code (next page):

(*Continued from previous page*)

```
ISR( TWI_vect )
{

    // The first step is to determine the reason the ISR got
    // 'triggered' in the first place:
    switch( I2C_SLVE_GetMasterReq() )
    {

        case I2C_STAT_MASTER_WANTS_TO_RECV:

            // ... do something involving 'I2C_SLVE_send()' ...

        break;

        case I2C_STAT_MASTER_WANTS_TO_SEND:

            // ... do something involving 'I2C_SLVE_get()' ...

        break;

        case I2C_STAT_MASTER_IS_FINISHED:

            // Finish the current transaction and keep the slave
            // enabled in case it is addressed again by the master.
            I2C_SLVE_finished( TRUE );

        break;

        default:

            /* ... Error code here ... */ ;

    } // end switch()

} // end ISR()
```

The very first task upon entering the ISR is to determine the reason the ISR was triggered in the first place. You do this by invoking `I2C_SLVE_GetMasterReq()`, which returns a value of type `I2C_STATUS`. This value will take on one of the three possible 'switch cases' outlined in the above snippet. If the master wants to 'receive', obviously, you have to *send* it some data; and if it wants to 'send', you must *get* some data, etc.

If the master is finished, then so must you, by invoking `I2C_SLVE_finished()`. The argument of 'TRUE' simply means that after completion of the communication transaction, the slave will continue to 'listen' for the *next* communication transaction initiation by the master. If it is set to 'FALSE', not only are you finishing the current communication transaction, but you're also ignoring any future ones – you're pulling yourself out of the I2C bus and no longer wishing to acknowledge any 'master' requests.

Running code in *interrupt context* can bring about other issues. If your code takes too long in processing, you'll delay other higher-priority interrupt tasks that are also running in the background and it'll become noticeable when your CEENBoT seems to act erratically, particularly when moving around. The best advice is that when writing 'slave' I2C code, to keep the code in the ISR short and sweet. Save the data in a buffer, but DO NOT process it inside of the ISR. When the ISR finishes, then another function, perhaps, outside of interrupt context, can then process the contents of the buffer and take action.

Function List Summary

- `I2C_open()` – Opens the I2C subsystem module for use.
- `I2C_close()` – Closes the I2C subsystem module after use.
- `I2C_set_BRG()` – Sets the I2C's *bit rate generator* value needed to obtain the desired operating clock rate frequency for the SCL line (*serial clock* line).
- `I2C_enable()` – Enables the I2C subsystem after it has been opened.
- `I2C_disable()` – Disables the I2C subsystem.
- `I2C_pullup_enable()` – Enables the *internal* pull-up resistors on the I2C bus.
- `I2C_pullup_disable()` – Disables the *internal* pull-up resistors on the I2C bus.
- `I2C_MSTR_start()` – Used to initiate a communication transaction as “master”.
- `I2C_MSTR_send()` – Used to transmit data to a slave device as “master”.
- `I2C_MSTR_send_multiple()` – Used to transmit a sequence of bytes to a slave device as “master”.
- `I2C_MSTR_get()` – Used to receive data from a slave device as “master”.
- `I2C_MSTR_get_multiple()` – Used to receive multiple bytes of data from a slave as “master”.
- `I2C_MSTR_stop()` – Used to complete a communication transaction with a slave as “master”.
- `I2C_SLVE_enable()` – Enables the I2C device for reception as slave device with assigned address.
- `I2C_SLVE_disable()` – Disables the ability of the I2C slave device to respond to its assigned address.
- `I2C_SLVE_get()` – Receive data from a master device as “slave”.
- `I2C_SLVE_send()` – Transmit data to a master device as “slave”.
- `I2C_SLVE_finished()` – Used to complete a communication transaction with a master device.
- `I2C_SLVE_GetMasterReq()` – Used to determine the reason the “master” device has addressed the “slave” device, so that the appropriate 'action' can be taken.
- `I2C_IsBusy()` – Used to determine if the I2C device (whether master or slave) is in the middle of a communication transaction.

Function Reference: Generic Functions

The `I2C_open()` Function

Format:

```
SUBSYS_OPENSTAT I2C_open( void )
```

Description:

Function acquires and initializes resources needed for operation of the **I2C**. You *must* call this function first with a successful 'open' before invoking any other function provided by this module.

Returns:

Returns a structure of type `SUBSYS_OPENSTAT` whose field entries indicate the status of the open request and the subsystem that resulted in an error (when, and *if* an error occurs). See *Chapter 1, Procedure for Opening Modules Before Use* for examples on how to handle the *open status*.

Example:

```
#include "capi324v221.h"

void CBOT_main( void )
{

    SUBSYS_OPENSTAT opstat;

    // Open the I2C module.
    opstat = I2C_open();

    if ( opstat.state == SUBSYS_OPEN )
    {

        // ... DO I2C STUF ...

    } // end if()

} // end CBOT_main()
```

The `I2C_close()` Function

Format:

```
void I2C_close( void )
```

Description:

Function deallocates and releases resources being used by the **I2C** subsystem module. No other functions should be invoked from this module once the subsystem module is closed.

The `I2C_set_BRG()` Function

Format:

```
void I2C_set_BRG( unsigned char bit_rate, I2C_PRESCALER prescaler )
```

Description:

Function allows the user to set the bit-rate value and prescaler value of the *bit-rate generator* (BRG) that is part of the I2C peripheral on the MCU needed to obtain the desired operating clock rate of the I2C. By default the **I2C** subsystem is set with am default *bit rate* and *prescaler* values to achieve the traditional **100kHz** clock rate common in many I2C devices. However, should you wish to attain a different clock rate (because the slave device you're trying to access uses a different clock rate), then you have to invoke this function immediately after `I2C_open()` to set the appropriate clock rate seen in the **SCL** line of the I2C-bus.

The `bit_rate` value and the *prescaler* value (denoted below by p) must be chosen according to the following equation:

$$\text{bit_rate} = \frac{\left(\frac{f_{\text{cpu}}}{f_{\text{SCL}}}\right) - 16}{2^{(2p+1)}} ,$$

where p is the *prescaler* value, which must be either: **1**, **4**, **16**, or **64**.; f_{cpu} is the frequency at which the MCU is running (presently **20MHz** – or **20x10⁶** to be precise); and f_{SCL} is the *target* clock rate we're trying to achieve on the **SCL** line of the I2C bus. The resulting bit_rate value is what you supply to the `I2C_set_BRG()` function along with your chosen *prescaler* value, provided as a predefined constant (see "*Input Arguments*" below).

Input Arguments:

`bit_rate` – You must supply to this argument, the value obtained from the above equation (see *Example* section below).

`prescaler` – You must supply to this argument, one of the following enumerated constants:

`I2C_PRESCALER_1`	– For a prescaler value of 1.
`I2C_PRESCALER_4`	– For a prescaler value of 4.
`I2C_PRESCALER_16`	– For a prescaler value of 16.
`I2C_PRESCALER_64`	– For a prescaler value of 64.

Example:

The following example, shows how to setup the bit-rate generator to obtain the standard 100kHz clock rate on the SCL line. We use the above equation with the following values: $f_{\text{cpu}} = 20\text{x}10^6$, $f_{\text{SCL}} = 100\text{x}10^3$, and $p = 1$. After plugging these values in the above equation we get a bit rate of 23 . Therefore, we would do something like:

(*Continued on next page*)

(*Continued from previous page*)

```
// Open the I2C subsystem.
I2C_open();

// Setup the bit-rate generator to achieve 100KHz on SCL.
I2C_set_BRG( 23, I2C_PRESCALER_1 );

// Get to work...
```

> **Note:** The above example is presented as an illustration only. Remember when you open the **I2C** subsystem module, it is already set to run at **100KHz** by default. You only have to perform the above procedure when you're trying to set the clock rate to something *other* than **100KHz**.

The `I2C_enable()` Function

Format:

```
void I2C_enable( void )
```

Description:

Function enables the **I2C** subsystem on the MCU. Note that when invoking `I2C_open()` for the *first time*, the **I2C** subsystem module is automatically enabled. This function is meant to be used if a prior `I2C_disable()` was issued for whatever reason. See '`I2C_disable()`' example section.

The `I2C_disable()` Function

Format:

```
void I2C_disable( void )
```

Description:

Function disables the I2C subsystem on the MCU. This function might be useful to temporarily disable the I2C subsystem, for example, if having a need to change the *bit rate generator* (BRG) values.

Example:

```
// ... where doing something happily, but now we need to change the clock rate.

// Disable the I2C.
I2C_disable();

// Set the new BRG values to achieve 50kHz operation.
I2C_set_BRG( 48, I2C_PRESCALER_1 );

// Enable the I2C.
I2C_enable();
```

The `I2C_pullup_enable() / I2C_pullup_disable()` Function

Format:

```
void I2C_pullup_enable( void )
void I2C_pullup_disable( void )
```

Description:

These functions can be used to effectively attach or detach the *internal* pullup resistors onto the I2C bus. The I2C specification requires that the I2C-bus lines be tied to a positive power supply as the lines are driven by open-collector/open-drain devices. These functions are provided for convenience for those 'simple' cases where the *internal* pullup resistors will do. However, for slightly heavier 'loads' on the I2C-bus, you may need to supply your own *external* pullup resistors. Please make sure you read the introductory section *"Other Hardware Requirements*" at the beginning of *this* chapter to better understand the need for the pullup resistors.

Example:

IF... you need to use the *internal* pullup resistors – you typically call this function soon after you open the I2C subsystem module via I2C_open():

```
// Open the I2C subsystem.
I2C_open();

// Enable the pullups.
I2C_pullup_enable();
```

Function Reference: Master-Mode Functions

The functions that follow are specifically used when the MCU is operating in *master* mode – this is the most common usage scenario when using I2C.

The `I2C_MSTR_start()` Function

Format:

```
I2C_STATUS I2C_MSTR_start( unsigned char slave_addr, I2C_MODE mode )
```

Description:

This function is used to *initiate* a communication transaction as “master” with a slave device on the I2C bus denoted by the specified slave address. It is the first function that must be invoked in order to begin the communication transaction. This function must be followed by one or more calls to `I2C_MSTR_send()`, or `I2C_MSTR_get()`, depending on the operating mode. You must then conclude the transaction with `I2C_MSTR_stop()`.

Input Arguments:

`slave_addr` – This is the slave address of the device you wish to begin a communication transaction with. Normally, the slave devices have their addresses predetermined by the manufacturer of the device and are, thus, traditionally of *fixed* value.

`mode` – You must specify the *directionality* of the communication between the master device or the slave device. You must provide a value of type `I2C_MODE`, which must be one of the following:

`I2C_MODE_MT` – To initiate a communication transaction as *master transmitter*.
`I2C_MODE_MR` – To initiate a communication transaction as *master receiver*.

Returns:

Function returns a value of type `I2C_STATUS`, which can take on the following predefined enumerated constants:

`I2C_STAT_OK` – No problems occurred.

`I2C_STAT_START_ERROR` – There was an error starting the communication transaction.

`I2C_STAT_ACK_ERROR` – The device refused to acknowledge when acknowledgment was expected.

`I2C_STAT_CONFIG_ERROR` – A 'START' was NOT possible because you have not configured the *bit rate generator* (BRG) via the '`I2C_set_BRG()`' function. Note that this may not necessarily be an issue because the BRG is set up for **`100kHz`** operation when first opened by default.

Example:

Please see the sample code *snippet* given in page **209**, in the beginning of this chapter.

The `I2C_MSTR_send()` Function

Format:

```
I2C_STATUS I2C_MSTR_send( unsigned char data )
```

Description:

Function can be used to send data [to a slave device] once a communication transaction has been started with a slave device on the bus with a prior call to '`I2C_MSTR_start()`' function. Multiple bytes can be sent to the slave device by simply invoking this function multiple times. Alternatively, you may want to consider using the `I2C_MSTR_send_multiple()` function, which can be used specifically for this purpose.

At some point of the communication transaction, you must either issue another `I2C_MSTR_start()`, changing the directionality of the data transmission, or complete the transaction by issuing a `I2C_MSTR_stop()`.

Input Arguments:

`data` – The *data byte* to transmit to the slave device in the current communication transaction.

Returns:

Function returns a value of type `I2C_STATUS`, which can take on the following predefined enumerated constants:

`I2C_STAT_OK` – No problems occurred.

`I2C_STAT_NO_ACK` – Device refused to acknowledge when acknowledgment was expected. This means the slave device did not acknowledge the last data byte sent. Whether is is considered an error or not depends on context. The slave may have received the byte but did not acknowledge on purpose to let you know its buffer is FULL and it cannot accept any more data. Alternatively, it could probably mean the slave did not receive the data after all. It is up to the user to determine how to handle this particular status, when encountered.

`I2C_STAT_NOT_MASTER` – This status is returned when you invoke this function accidentally as *slave* device, instead of *master* device. For example, you forgot to call `I2C_MSTR_start()` first. This should be considered an error.

`I2C_STAT_CONFIG_ERROR` – A *'Send'* was NOT possible because you have not configured the *bit rate generator* (BRG) via the '`I2C_set_BRG()`' function. Note that this may not necessarily be an issue because the BRG is set up for **`100kHz`** operation when first opened by default.

`I2C_STAT_UNKNOWN_ERROR` – The name says it all. Reason? *Unknown*.

Example:

Please see the sample code *snippet* given in page **209**, in the beginning of this chapter.

The `I2C_MSTR_send_multiple()` Function

Format:

```
I2C_STATUS I2C_MSTR_send_multiple( unsigned char *pBuffer, unsigned short int count )
```

Description:

This function works *exactly* like `I2C_MSTR_send()`, except it is used to send multiple bytes from a buffer back-to-back.

Input Arguments:

`pBuffer` – You must pass to this argument THE ADDRESS OF the buffer containing your data.

`count` – This argument specifies the number of bytes to read from the buffer and send out to the slave. It is important that this value is not greater than the size of the buffer itself. Failure to observe this rule will result in unpredictable behavior.

Returns:

Function returns a value of type `I2C_STATUS`, which can take on the following predefined enumerated constants:

`I2C_STAT_OK` – No problems occurred.

`I2C_STAT_NO_ACK` – Device refused to acknowledge when acknowledgment was expected. This means the slave device did not acknowledge the last data byte sent. Whether is is considered an error or not depends on context. The slave may have received the byte but did not acknowledge on purpose to let you know its buffer is FULL and it cannot accept any more data. Alternatively, it could probably mean the slave did not receive the data after all. It is up to the user to determine how to handle this particular status, when encountered.

`I2C_STAT_NOT_MASTER` – This status is returned when you invoke this function accidentally as *slave* device, instead of *master* device. For example, you forgot to call `I2C_MSTR_start()` first. This should be considered an error.

`I2C_STAT_CONFIG_ERROR` – A '*Send*' was NOT possible because you have not configured the *bit rate generator* (BRG) via the '`I2C_set_BRG()`' function. Note that this may not necessarily be an issue because the BRG is set up for **`100kHz`** operation when first opened by default.

`I2C_STAT_UNKNOWN_ERROR` – The name says it all. Reason? *Unknown*.

Example:

See the example for `I2C_MSTR_stop()` for a comprehensive example.

The `I2C_MSTR_get()` Function

Format:

```
I2C_STATUS I2C_MSTR_get( unsigned char *pData, BOOL acknowledge )
```

Description:

Function can be used to read or get data [from a slave device] once a communication transaction has been started with a slave device on the bus with a prior call to '`I2C_MSTR_start()`' function. Multiple bytes can be read from the slave device by simply invoking this function multiple times. Alternatively, you may want to consider using the `I2C_MSTR_get_multiple()` function, which can be used specifically for this purpose.

As per required by the I2C protocol, you have to acknowledge the slave of each byte received. This is what the second parameter is for. You must do this for every byte received except for the LAST one. This tells the slave device that you're about to STOP – or alternatively – to let the slave device know that you can't receive any more bytes because your internal buffer may be full.

At some point of the communication transaction, you must either issue another `I2C_MSTR_start()`, changing the directionality of the data transmission, or complete the transaction by issuing a `I2C_MSTR_stop()`.

Input Arguments:

`pData` – You must pass to this argument THE ADDRESS OF the variable where the received data will be written to.

`acknowledge` – You must specify 'TRUE' for this parameter each time you receive a byte to acknowledge to the slave that you have received the data. You do this except for the LAST byte received! (As dictated by the I2C protocol).

Returns:

`I2C_STAT_OK` – No problems occurred.

`I2C_STAT_NO_ACK` – Device refused to acknowledge when acknowledgment was expected. This means the slave device did not acknowledge the last data byte sent. Whether is is considered an error or not depends on context. The slave may have received the byte but did not acknowledge on purpose to let you know its buffer is FULL and it cannot accept any more data. Alternatively, it could probably mean the slave did not receive the data after all. It is up to the user to determine how to handle this particular status, when encountered.

`I2C_STAT_NOT_MASTER` – This status is returned when you invoke this function accidentally as *slave* device, instead of *master* device. For example, you forgot to call `I2C_MSTR_start()` first. This should be considered an error.

`I2C_STAT_CONFIG_ERROR` – A '*Get*' was NOT possible because you have not configured the *bit rate generator* (BRG) via the '`I2C_set_BRG()`' function. Note that this may not necessarily be an issue because the BRG is set up for **`100kHz`** operation when first opened by default.

`I2C_STAT_UNKNOWN_ERROR` – The name says it all. Reason? *Unknown*.

(*Continued from previous page*)

Example:

Please see the sample code *snippet* given in page **209**, in the beggining of this chapter.

The `I2C_MSTR_get_multiple()` Function

Format:

```
I2C_STATUS I2C_MSTR_get_multiple( unsigned char *pBuffer, unsigned short int count )
```

Description:

This function performs the same task as `I2C_MSTR_get()`, except it can be used to read multiple bytes back-to-back with a single call into a user-defined buffer. Also, since the function is told the exact number of bytes to transmit, the acknowledgment is performed automatically, with the last byte being NOT acknowledged, as required by the I2C protocol. (See `I2C_MSTR_get()` for details).

Input Arguments:

`pBuffer` – You must pass to this argument THE ADDRESS OF the buffer where data will be written to.

`count` – This argument specifies the number of bytes to read from the slave into the buffer. It is important that this value is not greater than the size of the buffer itself. Failure to observe this rule will result in unpredictable behavior.

Returns:

`I2C_STAT_OK` – No problems occurred.

`I2C_STAT_NO_ACK` – Device refused to acknowledge when acknowledgment was expected. This means the slave device did not acknowledge the last data byte sent. Whether is is considered an error or not depends on context. The slave may have received the byte but did not acknowledge on purpose to let you know its buffer is FULL and it cannot accept any more data. Alternatively, it could probably mean the slave did not receive the data after all. It is up to the user to determine how to handle this particular status, when encountered.

`I2C_STAT_NOT_MASTER` – This status is returned when you invoke this function accidentally as *slave* device, instead of *master* device. For example, you forgot to call `I2C_MSTR_start()` first. This should be considered an error.

`I2C_STAT_CONFIG_ERROR` – A '*Get*' was NOT possible because you have not configured the *bit rate generator* (BRG) via the '`I2C_set_BRG()`' function. Note that this may not necessarily be an issue because the BRG is set up for **`100kHz`** operation when first opened by default.

`I2C_STAT_UNKNOWN_ERROR` – The name says it all. Reason? *Unknown*.

(*Continued on next page*)

(*Continued from previous page*)

Example:

See the example for `I2C_MSTR_stop()` for a comprehensive example.

The `I2C_MSTR_stop()` Function

Format:

```
I2C_STATUS I2C_MSTR_stop( void )
```

Description:

This function is used to graciously complete a communication transaction with a slave device, that was started by a prior invocation to `I2C_MSTR_start()`. This allows the transaction to be completed in order to begin a new one by re-invoking `I2C_MSTR_start()` once again, in the appropriate mode (*master transmitter* or *master receiver*). Calling this function results in the master relinquishing the I2C bus, allowing other 'masters' (if they exist on the bus) to take a turn and grab hold of it.

Returns:

Function returns a value of type `I2C_STATUS` with the following enumerated constants:

`I2C_STAT_OK`	– No problems occurred.
`I2C_STAT_NOT_MASTER`	– This status is returned when you invoke this function accidentally as *slave* device, instead of *master* device. For example, you forgot to call `I2C_MSTR_start()` first. This should be considered an error.

Example:

Please see the sample code *snippet* given in page **209**, in the beginning of this chapter.

Function Reference: Slave-Mode Functions

The following functions can only be used when operating in *slave* mode. IF, and when, invoked, they're normally placed inside of the `TWI_vect` ISR function as shown in the beginning of this chapter. Recall that *slave* mode is not an exclusive operation. It is possible for an I2C device to have both modes *enabled* – they just cannot be 'active' at the same time. That is, at any one time, an I2C device is a master of the bus (while everyone else is acting as *slave*), or alternatively, be forced to act as a 'slave', because another device is 'master' due to it being able to 'grab hold' of the I2C bus *first*.

In any case, responses to I2C events from 'another master' require that you perform any one of the following 'slave' operations accordingly.

The `I2C_SLVE_enable()` Function

Format:

```
I2C_STATUS I2C_SLVE_enable( unsigned char slave_addr )
```

Description:

This function activates the 'slave' portion of the I2C device of the MCU for reception. This function performs three tasks: a) First, it sets the slave address that the I2C device is going to respond to when in slave mode; b) Second, it enables the ability of the I2C to acknowledge its own address (when another master on the I2C-bus wishes to do so); c) Finally, it enables I2C interrupts so that the user's ISR (interrupt service routine) will be triggered when a 'slave operation' occurs.

Also, you *must* make sure you're not in the middle of a communication transaction when invoking this function. You should use the `I2C_IsBusy()` function for this purpose.

Input Arguments:

`slave_addr` – This is the address the *slave address* that you assign to yourself. The address that you will respond to when another 'master' on the bus wishes to talk to you. It can be any value between **0x01** and **0xFE**. Addresses **0x00** and **0xFF** are reserved – please do not use these addresses.

Returns:

Function returns a value of type `I2C_STATUS`, with one of the following enumerated constants:

`I2C_STAT_OK` – No problems occurred.

`I2C_STAT_BUSY` – This status code is returned if you attempt to disable the SLAVE device in the middle of a transaction, be it a MASTER or SLAVE.

Example:

Refer to the *slave* sample snippet code given on page **210**. Also see the example for `I2C_SLVE_disable()` function.

The `I2C_SLVE_disable()` Function

Format:

```
I2C_STATUS I2C_SLVE_disable( void )
```

Description:

This function disables the *slave* portion of the I2C device. That is, it disables the I2C's ability to respond to its previously assigned slave address when such attempts are made by another 'master' on the same I2C-bus. It also disables the generation of interrupts as a result of the same.

As with `I2C_SLVE_enable()`, you cannot call this function in the middle of a communication transaction. You must wait until any communication transactions complete before invoking this function. Use the `I2C_IsBusy()` function to determine this.

Returns:

Function returns a value of type `I2C_STATUS`, with one of the following enumerated constants:

- `I2C_STAT_OK` – No problems occurred.
- `I2C_STAT_BUSY` – This status code is returned if you attempt to disable the SLAVE device in the middle of a transaction, be it a MASTER or SLAVE.

Example:

```
// Initialization code (somewhere in your program).

// Open the I2C subsystem module.
I2C_open();

// Enable the slave portion of the I2C device and self-assign
// slave address of 0x3A.  We need to make sure we don't do this in the
// middle of a communication transaction.
while ( I2C_IsBusy() == TRUE );

    I2C_SLVE_enable( 0x3A );
```

The cleanup code...

```
// Disable the slave appropriately.
while( I2C_IsBusy() );  // Wait until the slave finishes (just in case).

// Disable it.
I2C_SLVE_disable();
```

The `I2C_SLVE_get()` Function

Format:

```
I2C_STATUS I2C_SLVE_get( unsigned char *pData, BOOL acknowledge )
```

Description:

This function is used to read data sent from a 'master' device on the I2C bus. This function must be used inside of your `TWI_vect` ISR. You must also ALWAYS acknowledge back for each byte received, unless there's some exceptional reason for you *not* to do so. In addition, you should always check the return status value of the function to determine if `I2C_STAT_START_STOP_REQ` is returned, at which point you *must* follow this function with a call to `I2C_SLVE_finished()`.

Input Arguments:

`pData` – You must pass to this argument THE ADDRESS OF a variable where the received data will be stored.

`acknowledge` – Should always be set to '`TRUE`' unless you wish to let the Master know that something went wrong, at which point, then it should be set to '`FALSE`'. There's technically, however, no reason *not* to acknowledge.

Returns:

Function returns a value of type `I2C_STATUS`, which can be one of the following enumerated constants:

`I2C_STAT_OK` – No problems occurred.

`I2C_STAT_MASTER_BUSY` – This status is returned when a slave operation is attempted when the device is in the middle of an operation in *master* mode.

`I2C_STAT_NO_ACK` – This status shows up when you requested a data byte from the master (and you got it), but *you* (the slave) refused to send an acknowledge back. In fact, this status code is an acknowledgment to you (the *slave*) that the master understood your refusal to acknowledge, because you specified '`FALSE`' to the '`acknowledge`' argument of the function.

`I2C_STAT_START_STOP_REQ` – This status is returned when the *master* I2C device on the bus has either completed the current communication transaction, or wishes to begin a new one. In any case, this status is returned to inform you that you should call `I2C_SLVE_finished()` immediately after.

`I2C_STAT_UKNOWN_ERROR` – Take a guess at what this means... ;o)

Example:

See the example for `I2C_GetMasterReq()` function for a quick example.

The `I2C_SLVE_send()` Function

Format:

```
I2C_STATUS I2C_SLVE_send( unsigned char data, BOOL acknowledge_expected )
```

Description:

Function is used to send data to a 'master' device on the I2C bus. This function must be used inside of your `TWI_vect` ISR. Moreover, you must expect an acknowledge for each data byte sent, except for the *last one*. This is the master's way of letting you know that it understands it has received the last byte.

Input Arguments:

`data` – The data byte to send to the master I2C device.

`acknowledge_expected` – You should expect the I2C master device to acknowledge the reception of *each* byte except for the last one. So, set this always to '`TRUE`' except for the last byte.

Returns:

Function returns a value of type `I2C_STATUS`, with a value consisting of one of the following enumerated constants:

`I2C_STAT_OK` – No problems occurred.

`I2C_STAT_MASTER_BUSY` – This status is returned when a slave operation is attempted when the device is in the middle of an operation in *master* mode.

`I2C_STAT_NO_ACK` – This status indicates that the *master* did NOT acknowledged the previous data byte sent. Now, whether this is considered an error or not depends on context. The master may have received the byte but did not acknowledge on purpose to let you know its buffer is full and it cannot accept any more data. Also, a NOT ACKNOWLEDGE can be expected when the master *knows* that it has received the last data byte in a multi-byte transmission sequence. Alternatively, it could probably mean that the master, after all, did *not* receive the data. It is up to the user to determine how to handle this particular status code.

`I2C_STAT_ACK_ERROR` – This status is returned when an acknowledgment error has occurred. This normally happens when a different acknowledgment other than what was expected has occurred. Unlike `I2C_STAT_NO_ACK`, this status *should* be considered an ERROR.

`I2C_STAT_UNKNOWN_ERROR` – Something beyond human comprehension has just occurred.

Example:

See the example for `I2C_GetMasterReq()` function for a quick example.

The `I2C_SLVE_finished()` Function

Format:

```
I2C_STATUS I2C_SLVE_finished( BOOL keep_enabled )
```

Description:

This function is used to complete a communication transaction in response to the master letting *you* know it has completed a transaction and is relinquishing the I2C-bus. This function must be used inside of your `TWI_vect` ISR. The function also gives the user the opportunity to disable the *slave* portion of the I2C, so that subsequent 'master' requests are ignored.

Input Arguments:

`keep_enabled` – If set to '`TRUE`', the *slave* I2C subsystem will remain active. If set to '`FALSE`' it will disable it, as if you had called `I2C_SLVE_disable()` directly. So it can work as two functions rolled into one.

Returns:

Function returns a status value of type `I2C_STATUS`, which can be any one of the following enumerated constants:

`I2C_STAT_OK` – No problems occurred.

`I2C_STAT_NOT_SLAVE` – This status was returned because the you invoked this function while the I2C device is operating as *master*.

Example:

See the example for `I2C_GetMasterReq()` function for a quick example.

The `I2C_SLVE_GetMasterReq()` Function

Format:

```
I2C_STATUS I2C_SLVE_GetMasterReq( void )
```

Description:

This function is the *most* important function when operating in *slave* mode. This function must be used inside of your `TWI_vect` ISR. It should be the *first* I2C function inside of your ISR, used to determine the reason the ISR was triggered in the first place – that is, to find out what it is exactly that the 'master' has requested from *you* – the slave device. The *Example* section for this function gives you a general idea on how this should be used.

(*Continued on next page*)

(*Continued from previous page*)

Returns:

Function returns a value of type `I2C_STATUS`, consisting of one of the following enumerated constants:

`I2C_STAT_MASTER_WANTS_TO_RECV` – This is an indication that the I2C device is expecting that you (the *slave*) send it some data. Naturally this means you will fallow this with `I2C_SLVE_send()` at some point.

`I2C_STAT_MASTER_WANTS_TO_SEND` – This is an indication that the I2C device wants to send you (the *slave*) some data. Naturally this means you will follow this with `I2C_SLVE_get()` at some point.

`I2C_STAT_MASTER_IS_FINISHED` – This is an indication that the master I2C device is finished with the current communication transaction and so should you (the *slave*). You must follow this with `I2C_SLVE_finished()` at some point.

`I2C_STAT_MASTER_UNKNOWN_REASON` – The 'intentions' of the *master* device cannot be determined. You should treat this as an ERROR.

Example:

The following example conveys the general idea on how you should use this function inside of your `TWI_vect` ISR function – note that this short snippet is not realistically useful and is only meant to convey the general idea you should take:

```
// The ISR:
ISR( TWI_vect )
{

    // Data buffer.
    static unsigned char data_buffer[ 16 ];
    static unsigned short int i = 0;
    I2C_STATUS status;

    // The first step is to determine the reason the ISR got
    // 'triggered' in the first place:
    switch( I2C_SLVE_GetMasterReq() )
    {

        case I2C_STAT_MASTER_WANTS_TO_RECV:

            // ... do something involving 'I2C_SLVE_send()' ...
            // For example:
            I2C_SLVE_send( 0x01, TRUE );
            I2C_SLVE_send( 0x02, TRUE );
            I2C_SLVE_send( 0x03, FALSE ); // Last byte.

        break;
```

(*Continued on next page*)

(*Continued from previous page*)

```
        case I2C_STAT_MASTER_WANTS_TO_SEND:

            // ... do something involving 'I2C_SLVE_get()' ...

            // For example -- store the bytes in a buffer
            // as they arrive.
            status = I2C_SLVE_get( &data_buffer[ i++ ], TRUE );

            if ( status == I2C_STAT_START_STOP_REQ )
            {

                // Reset the byte counter.
                i = 0;

                // Finish the transaction.
                I2C_SLVE_finished( TRUE );

            } // end if()


        break;

        case I2C_STAT_MASTER_IS_FINISHED:

            // Reset the byte counter.
            i = 0;

            // Finish the current transaction and keep the slave
            // enabled in case it is addressed again by the master.
            I2C_SLVE_finished( TRUE );

        break;

        default:

            /* ... Error code here ... */ ;

    } // end switch()

} // end ISR()
```

The `I2C_IsBusy()` Function

Format:

```
BOOL I2C_IsBusy( void )
```

Description:

Function can be used to check whether the I2C device is in the middle of a communication transaction either as *master* or *slave*. The function is particularly useful when enabling or disabling either the master device or the slave device of the I2C.

Returns:

Function returns a value of type `BOOL`, being:

`TRUE` – If the I2C device is in the middle of a communication transaction.
`FALSE` – If the I2C is *not* in the middle of a communication transaction.

Example:

Suppose we need to change the baud rate some time *later* from the default 100KHz to 50KHz:

```
// Wait for the current communication transaction to finish - if any:
while( I2C_IsBusy() );

// Disable the I2C.
I2C_disable();

// Set the new bit-rate values.
I2C_set_BRG( 48, I2C_PRESCALER_1 );

// Enable it back on.
I2C_enable();

// Back in business. :o)
```

Chapter 17: Helper Utilities

This chapter covers some fundamental 'helper' utilities available 'across the board' on the API. Presently many of these helper utilities focus on the manipulation of I/O Port bits.

Module at a Glance

Description

The file **`utils324vXXX.h`** (which is automatically included when you include **`capi324vXXX.h`**) defines some useful *helper* macros that can be used for manipulating both the direction and state of I/O port bits.

Modular Dependencies

None whatsoever. Helper macros from the *utilities* header file are always available for use.

Hardware Dependencies

None.

Macro List Summary

Port Bit Manipulation Macros

- `GBV()` – Stands for *Get Bit Value.* Used to obtain the state of an I/O port bit.
- `SBV()` – Stands for *Set Bit Value*. Used to set the state of an I/O port bit to "`1`".
- `CBV()` – Stands for *Clear Bit Value*. Used to clear the state of I/O port bit to "`0`".
- `TBV()` – Stands for *Toggle Bit Value*. Used to toggle the state of an I/O port bit from "`0`" to "`1`" and vice versa.
- `SBD()` – Stands for *Set Bit Direction*. Used to set the I/O port bit direction.

Delay-based Macros

- `DELAY_ms()` – Macro for *'blocking'* delay in milliseconds. Can be disabled when '`__NO_DELAYS`' is defined when debugging or simulating.
- `DELAY_us()` – Macro for '*blocking*' delay in microseconds. Can be disabled when '`__NO_DELAYS`' is defined when debugging or simulating.

Function/Macro Reference

Port Bit Manipulation Macros

The `GBV()` Macro

Format:

```
GBV( b, port )
```

Description:

`GBV()` stands for *Get Bit Value.* This macro can be used to obtain the state of the bit number denoted by *`b`* of a given *`port`* address. See 'Example' below for usage.

Input Arguments:

b – The bit number whose state you're requesting. Bit numbers start from zero!

port – The port being accessed, must be one of the following:

`PINA` – For accessing **`Port A`** when the bit in question is configured as *input*.
`PINB` – For accessing **`Port B`** when the bit in question is configured as *input.*
`PINC` – For accessing **`Port C`** when the bit in question is configured as *input.*
`PIND` – For accessing **`Port D`** when the bit in question is configured as *input.*

Or...

`PORTA` – For accessing **`Port A`** when the bit in question is configured as *output*.
`PORTB` – For accessing **`Port B`** when the bit in question is configured as *output.*
`PORTC` – For accessing **`Port C`** when the bit in question is configured as *output.*
`PORTD` – For accessing **`Port D`** when the bit in question is configured as *output.*

Example:

Suppose we're monitoring **`PA7`**, which has been configured as *input*, and we're waiting for it to go high for whatever reason, and once we do we take action by invoking the phony function `take_action()`:

```
// Somewhere in 'CBOT_main()'

while( ! ( GBV( 7, PINA ) );    // Wait here while PA7 is LOW.
                                // Note the inversion '!'.

// If we get to this point it's because PA7 has gone HIGH.
// Double check and take action.  If HIGH...
if ( GBV( 7, PINA ) )

    // Take action!
    take_action();
```

Now, suppose PA7 was configured as an *output* pin instead. Then we would have used `GBV( 7, PORTA )`.

The `SBV()` Macro

Format:

```
SBV( b, port )
```

Description:

`SBV()` stands for *Set Bit Value*. It can be used to *set* the state of an I/O port bit to "`1`" (set). In addition, if the corresponding I/O port bit in question was actually configured as an *input pin* instead of an *output pin*, then you'll end up manipulating the status of the *pull-up* resistor (attach "`1`" (set) or detach "`0`" (clear) from port bit).

Input Arguments:

b – The bit number whose state you're requesting. Bit numbers start from zero!

port – The port being accessed, must be one of the following:

- `PORTA` – For manipulating **`Port A`**.
- `PORTB` – For manipulating **`Port B`**.
- `PORTC` – For manipulating **`Port C`**.
- `PORTD` – For manipulating **`Port D`**.

Example:

Suppose **`Port B`** has all of its bits cleared to zero (from a prior hardware RESET). Furthermore suppose we need to have bits **`PB5`** and **`PB2`** of **`Port B`** set to **`1`**:

```
// Somewhere in 'CBOT_main()'

SBV( 5, PORTB );    // Set PB5.
SBV( 2, PORTB );    // Set PB2.
```

The `CBV()` Macro

Format:

```
CBV( b, port )
```

Description:

`CBV()` stands for *Clear Bit Value*. It can be used to *clear* the state of an I/O port bit to "`0`" (cleared).

(*Continued on next page*)

Input Arguments:

b – The bit number whose state you're requesting. Bit numbers start from zero!

port – The port being accessed, must be one of the following:

PORTA – For manipulating **Port A**.
PORTB – For manipulating **Port B**.
PORTC – For manipulating **Port C**.
PORTD – For manipulating **Port D**.

Example:

```
// Somewhere in 'CBOT_main()'

CBV( 5, PORTB );    // Clear PB5.
CBV( 2, PORTA );    // Clear PA2.
```

The TBV() Macro

Format:

TBV(*n*, *port*)

Description:

TBV() stands for *Toggle Bit Value*. It can be used to toggle the state of an I/O port bit from "0" to "1" and vice versa.

Input Arguments:

b – The bit number whose state you're requesting. Bit numbers start from zero!

port – The port being accessed, must be one of the following:

PORTA – For manipulating **Port A**.
PORTB – For manipulating **Port B**.
PORTC – For manipulating **Port C**.
PORTD – For manipulating **Port D**.

(*Continued on next page*)

(*Continued from previous page*)

Example:

Suppose we have an LED connected to **PA6**. We can toggle this LED on and off as follows:

```
// Somehwere in 'CBOT_main()'...

while( 1 )
{

    TBV( 6, PORTA );    // Toggle PA6.

    DELAY_ms( 250 );    // Wait a bit before doing it again...

}
```

The `SBD()` Macro

Format:

`SBD(` *`ddr_port`*`,` *`b`*`,` *`dir`* `)`

Description:

`SBD()` stands for *Set Bit Direction*. It is used to set the directionality of an I/O port bit.

Input Arguments:

`ddr_port` – The *data direction register* whose port bit we're manipulating. It needs to be one of the following:

- `A` – For manipulating the directionality of **`Port A`**.
- `B` – For manipulating the directionality of **`Port B`**.
- `C` – For manipulating the directionality of **`Port C`**.
- `D` – For manipulating the directionality of **`Port D`**.

> **Note:** Notice these are *single letters*. DO NOT refer to the actual data-direction register designation, such as **`DDRA`**, **`DDRB`**, etc, when using `SBD()` macro.

`b` – The bit number whose state you're requesting. Bit numbers start from zero!

`dir` – The directionality of the specified port bit. It must be one of the following:

- `INPIN` – To configure the I/O port bit as *input pin*.
- `OUTPIN` – To configure the I/O port bit as *output pin*.

(*Continued on next page*)

(*Continued from previous page*)

Example:

Suppose we need to set **PA7**, **PA6** and **PA5** as outputs, and **PA4**, **PA3** as inputs.

```
// Somehwere in 'CBOT_main()'...

SBD( A, 7, OUTPIN );    // PA7 as OUTPUT.
SBD( A, 6, OUTPIN );    // PA6 as OUTPUT.
SBD( A, 5, OUTPIN );    // PA5 as OUTPUT.

SBD( A, 4, INPIN );     // PA4 as INPUT.
SBD( A, 3, INPIN );     // PA3 as INPUT.

// Make sure the output in are LOW.
CBV( 7, PORTA );
CBV( 6, PORTA );
CBV( 5, PORTA );

// Enable the PULL-UP resistor for PA3.
SBV( 4, PORTA );
```

Note that, in the last line, because PA4 was configured as an *input pin*, that setting this bit via '`PORTA`' designator manipulates the state of the *pull-up resistor* instead. This 'behavior' is true in general across all four ports.

Delay-based Macros

The `DELAY_ms()`/`DELAY_us()` Macros

Format:

`DELAY_ms( t )` and `DELAY_us( t )`

Description:

These are 'blocking' delay macros that essentially invoke `_delay_ms()` and `_delay_us()` respectively. However, these delay macros have the benefit that they can be disabled when '`__NO_DELAYS`' is defined. This is useful when debugging and/or simulating where delays can 'hinder' this process.

Input Arguments:

t – This parameter specifies the *delay* quantity in units of *milliseconds* or *microseconds* respectively. Note that the maximum delay value is imposed by whatever `_delay_ms()` and `_delay_us()` delay limitations are in the AVR Lib-C library.

See: http://www.nongnu.org/avr-libc/user-manual/group__util__delay.html about this.

Example:

```
// Somewhere in 'CBOT_main()'...

// Uncomment when debugging.
// #define __NO_DELAYS

while( 1 )
{

    // Toggle the LED.
    LED_toggle( LED_Green );

    // Wait....
    DELAY_ms( 100 );

}
```

In the above example, we have a simple loop, where we toggle the green LED every **`100ms`**. If we then choose to debug or simulate we can uncomment the "`#define`" above so that "`#define __NO_DELAYS`" becomes active. This essentially disables `DELAY_ms()` into a *no-operation*.