Chapter 5a: PSoC 6 / AnyCloud Core Concepts

After completing this chapter, you will understand what the PSoC solution is, what tools are included, and how to use those tools to compile and run programs on your device.

5a.1 PSoC 6 Solution Tour

The PSoC 6 solution & Reference flow contains a “classic” MCU development flow and it includes:

- The Eclipse IDE for ModusToolbox
- Hardware Abstraction Layer (HAL)
- Board Support Packages (BSPs) for Cypress kits
- Peripheral Driver Library (PDL)
- Libraries (Retarget-IO, emWin etc.)
- Programming and debug tools

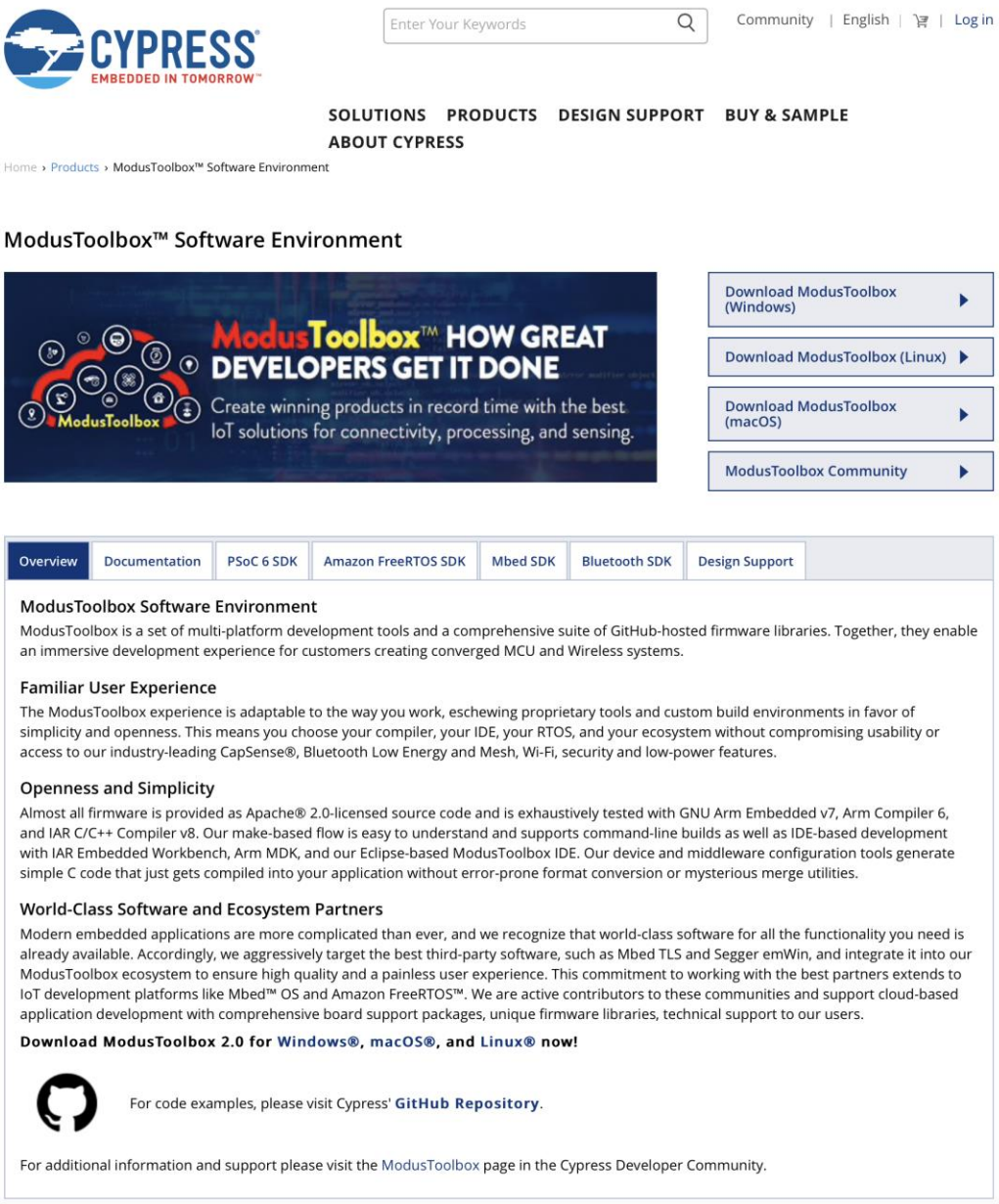
This section provides a brief tour of the ModusToolbox websites, Eclipse IDE, and various tools.

5a.1.1 Web

Cypress.com

On the Cypress website, you can download the software, view the documentation, and access the solutions:

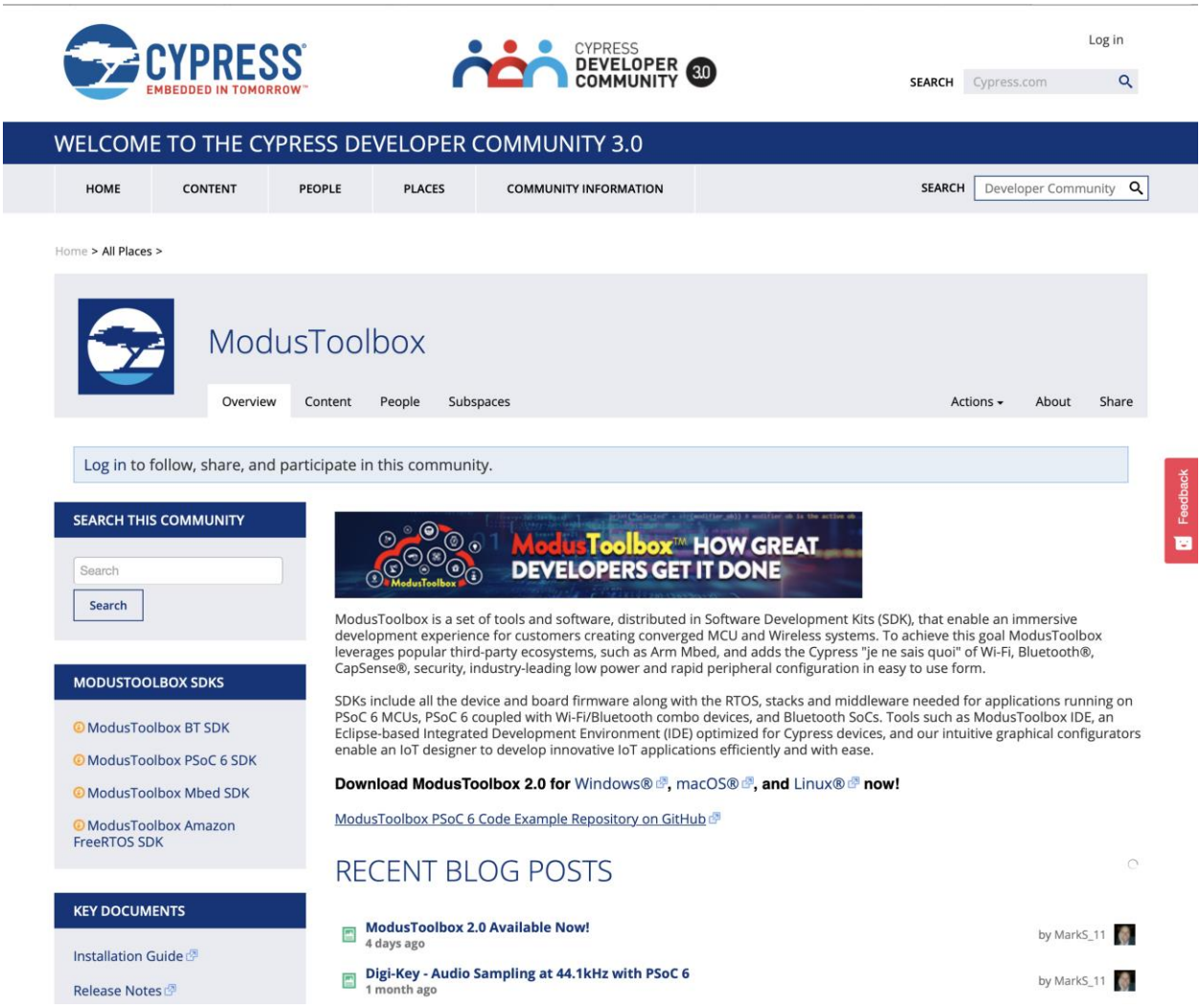
https://www.cypress.com/products/modustoolbox-software-environment



Community

On the ModusToolbox Community website, you can interact with other developers and access various Knowledge Base Articles (KBAs):

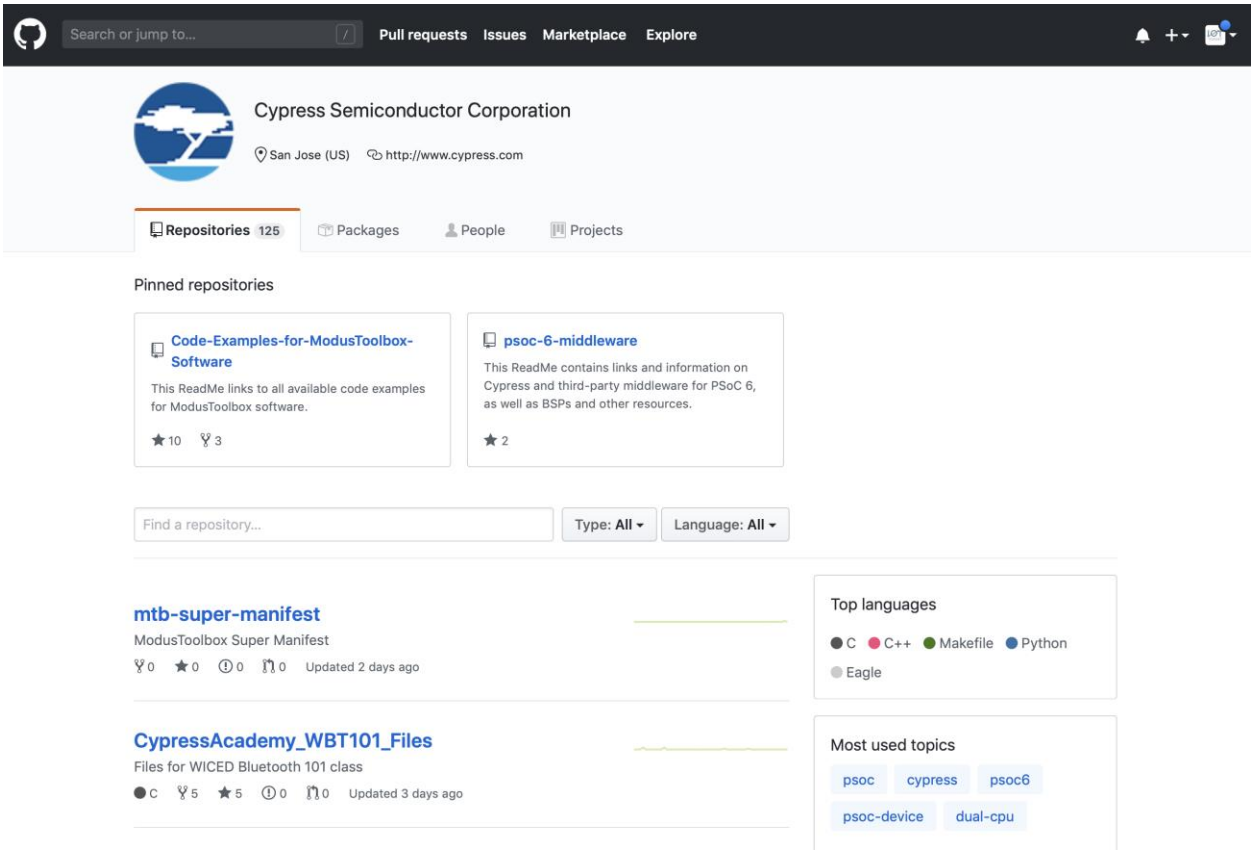
https://community.cypress.com/community/modustoolbox



Github.com

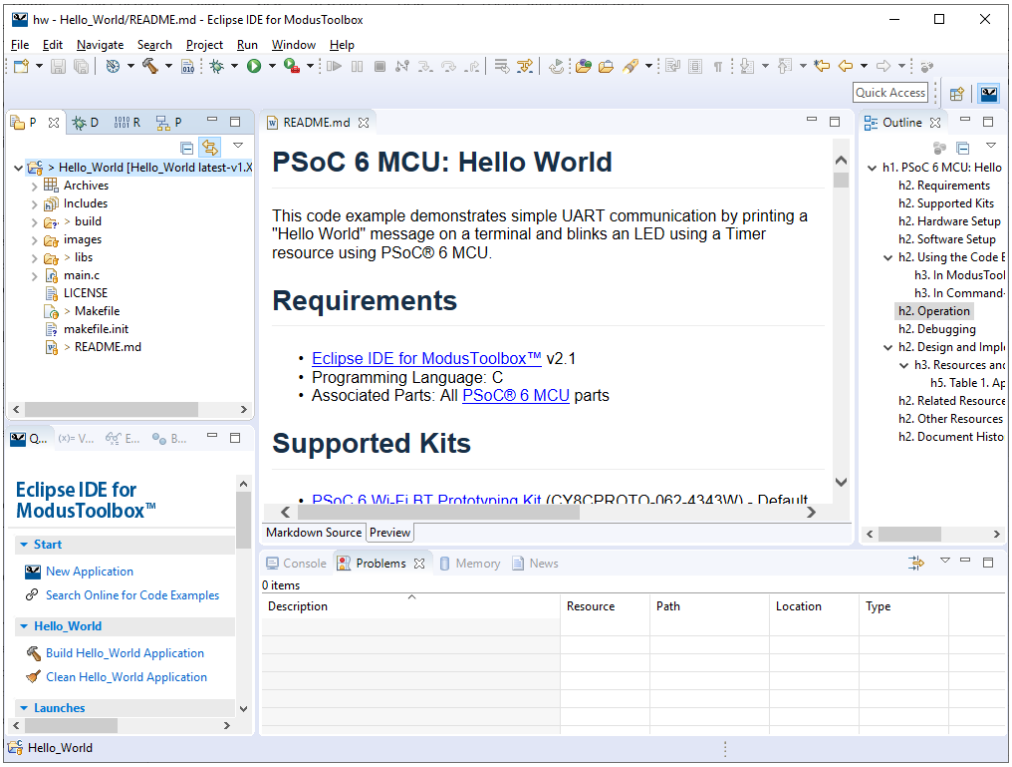
The Cypress GitHub website contains all the BSPs, code examples, and libraries for use with various ModusToolbox tools.

https://github.com/cypresssemiconductorco



5a.1.2 Eclipse-Based IDE

The Eclipse IDE for ModusToolbox uses several plugins, including the Eclipse C/C++ Development Tools (CDT) plugin. The IDE contains Eclipse standard menus and toolbars, plus various views such as the Project Explorer, Code Editor, and Console.



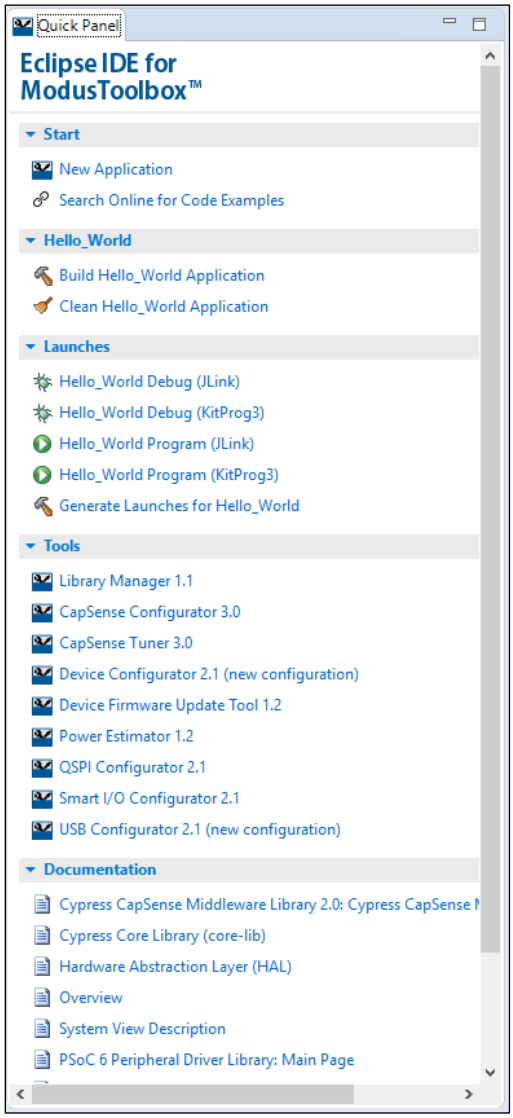
For more information about the IDE, refer to the "Eclipse IDE for ModusToolbox User Guide":

http://www.cypress.com/MTBEclipseIDEUserGuide

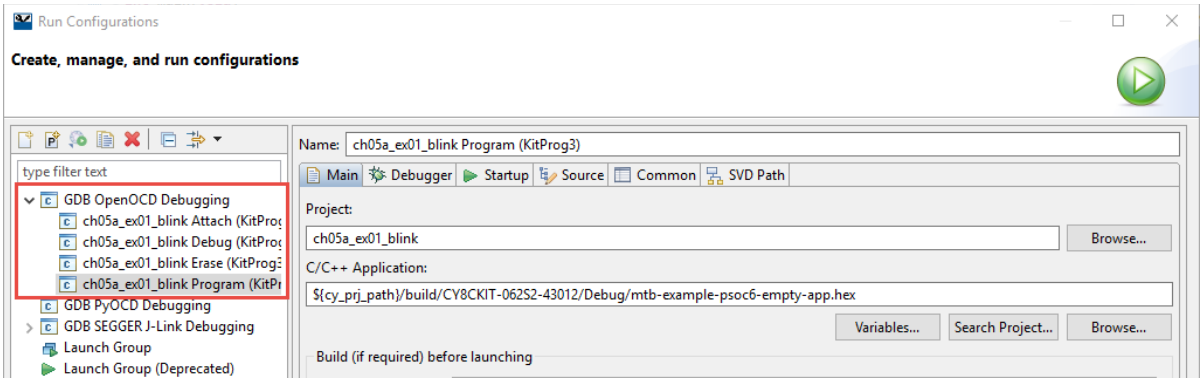
Quick Panel

Cypress has extended the Eclipse functionality with a "ModusToolbox" Perspective. Among other things (such as adding a bunch of useful Debug windows), this perspective includes a panel with links to commonly used functions including:

- Create a New Application
- Clean & Build
- Program/Debug Launches
- Tools (Configurators)
- Documentation

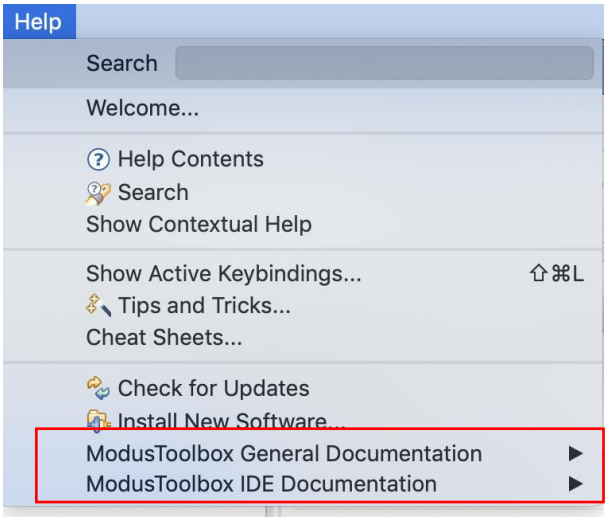


In addition to the launch configurations in the Quick Panel, there are usually others that you can find under **Run > Run Configurations > GDB OpenOCD Debugging**. These typically include configurations to erase the chip or attach the debugger to a running target. You can also see exactly what the other launch configurations do from that menu.



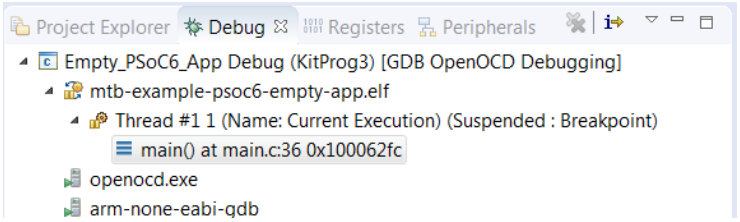
Help Menu

The IDE Help menu provides links to general documentation, such as the ModusToolbox User Guide and ModusToolbox Release Notes, as well as IDE-specific documentation, such as the Eclipse IDE for ModusToolbox Quick Start Guide.



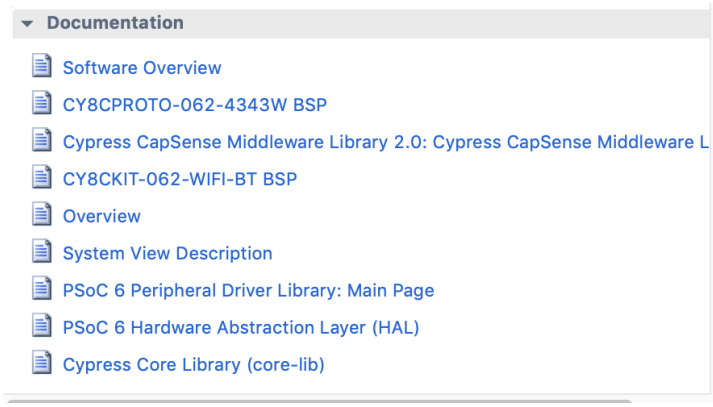
Integrated Debugger

The Eclipse IDE provides an integrated debugger using either the KitProg3 (OpenOCD) on the PSoC 6 development kit, or through a MinProg4 or Segger J-Link debug probe.



Documentation

After creating a project, assorted links to documentation are available directly from the Quick Panel, under the “Documentation” section.



Eclipse IDE Tips & Tricks

Eclipse has several quirks that new users may find hard to understand at first. Here are a few tips to make the experience less difficult:

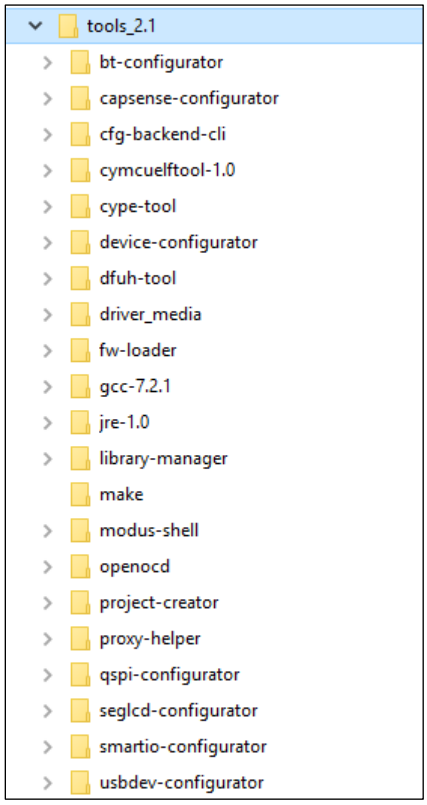
- If your code has IntelliSense issues, use **Program > Rebuild Index** and/or **Program > Build**.
- Sometimes when you import a project, you don't see all the files. Right-click on the project and select **Refresh**.
- Various menus and the Quick Panel show different things depending on what you select in the Project Explorer. Make sure you click on the project you want when trying to use various commands.
- Right-click in the column on the left- side of the text editor view to pop up a menu to:
    - Show/hide line numbers
    - Set/clear breakpoints

Refer also to the Cypress Eclipse Survival Guide for more tips and tricks.

5a.1.3 Tools

The ModusToolbox installer includes several tools that can be used along with the IDE, or as stand-alone tools. These tools include Configurators that are used to configure hardware blocks, as well as utilities to create projects without the IDE or to manage BSPs and libraries. All the tools can be found in the installation directory. The default path (Windows) is:

C:/Users/<user-name>/ModusToolbox/tools_2.1:



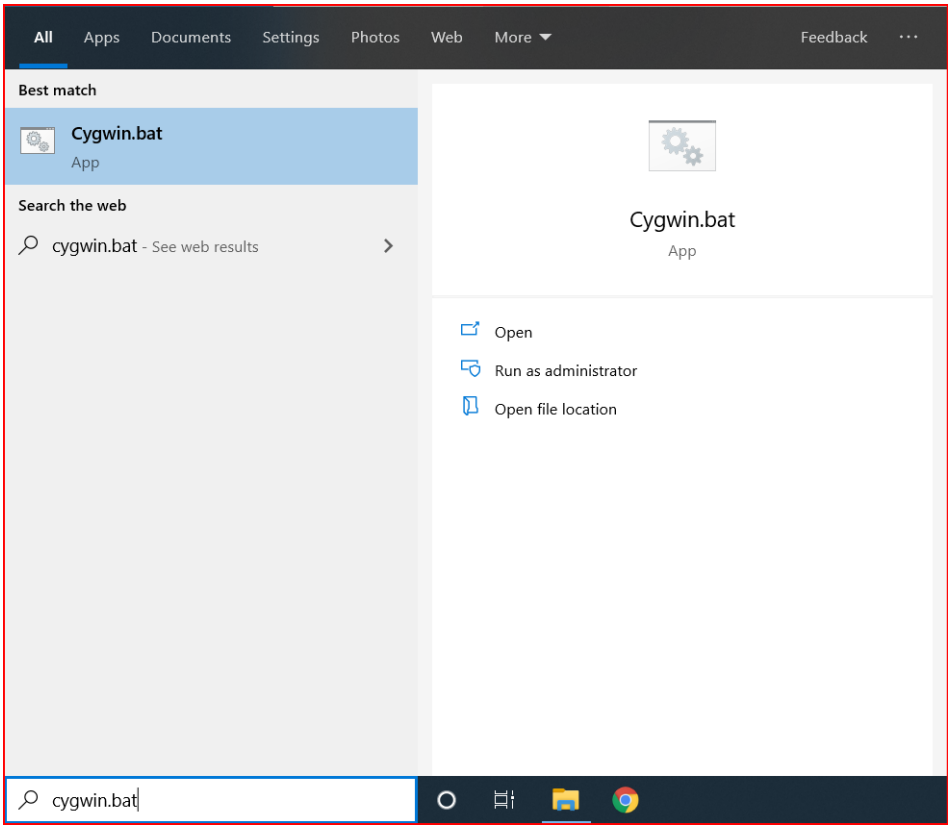
5a.1.4 Command Line

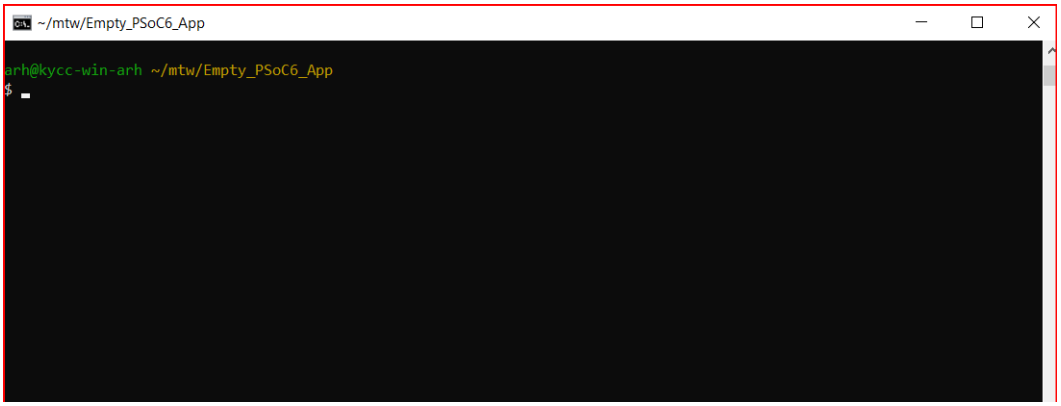
You can run all the tool functions using the command line.

Windows

To run the command line, you need a shell that has the correct tools (such as git, make, etc.). This could be your own Cygwin installation or Git Bash. ModusToolbox includes the “modus shell”, which is based on Cygwin. To run this, search for Cygwin.bat (it is in the ModusToolbox installation under ModusToolbox/tools_2.1/modus-shell).

Note: modus-shell is only expected to be required by developers who don't already have something like Cygwin installed.





macOS / Linux

To run the command line on macOS or Linux, just open a terminal window.

Make Targets (Commands)

In order to have the command line commands work, you need to be at the top level of a ModusToolbox project where the Makefile is located.

The following table lists the most commonly used make targets (i.e. commands). Refer also to the ModusToolbox User Guide document.

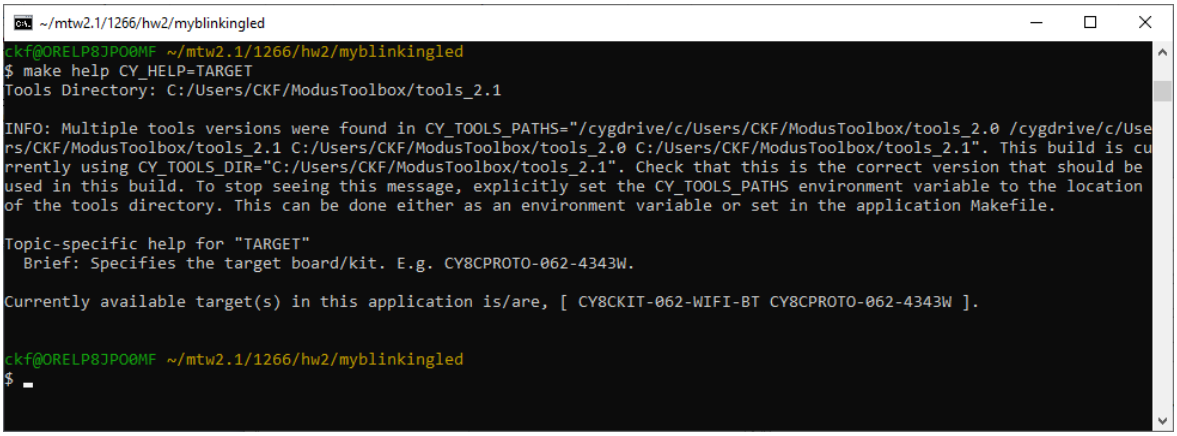
Make Command	Description
`make help`	This command will print out a list of all the make targets. To get help on a specific target type `make help CY_TARGET=getlibs` (or whichever target you want).
`make getlibs`	Hierarchically, process all the ".lib" files and bring all the libraries into your project.
`make debug`	Build your project, program it to the device, and then launch a GDB server for debugging.
`make program`	Build and program your project.
`make qprogram`	Program without building.
`make config`	This command will open the Device Configurator.

The help make target by itself will print out top level help information. For help on a specific variable or target use `make help CY_HELP=<variable or target>`. For example:

```
make help CY_HELP=build
```

or

```
make help CY_HELP=TARGET
```



Accessing Tools from the Command Line

There are make targets to launch some of the tools from an application's folder. Use `make help` and look for *Tools make targets* for a full list. A couple useful examples:

- `make config` launches the Device Configurator and opens the application's configuration file.
- `make modlib` launches the Library Manager and opens the application's library settings.

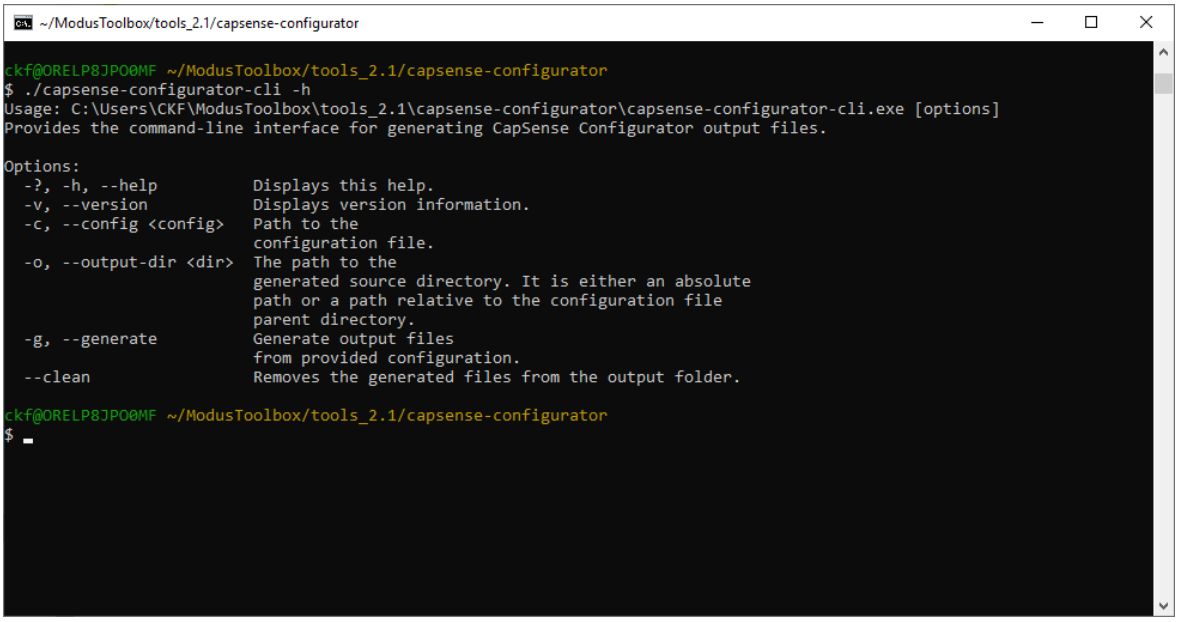
You can launch other ModusToolbox tools using `make open` and specifying a tool name or a file to open. Use `make help CY_HELP=open` for details. For example:

- `make open CY_OPEN_TYPE=capsense-tuner` launches the CapSense Tuner and opens the application's design.cycapsense file.

Hint, if you run `make open CY_OPEN_TYPE=junk` it displays a complete list of valid tool types.

All the tools can be run directly from the tools folder as well. Launch each tool from its respective directory inside ModusToolbox/tools_2.1. For many tools, there is a command line (i.e. non-GUI) version that can be useful for scripting/automation. The `-h` option shows help for the CLI tools. For example:

```
./capsense-configurator-cli -h
```



5a.2 Demo Walkthrough

5a.2.1 Blinking LED from the Eclipse IDE

Demonstration of how to create a project, build it, and program the kit.

5a.2.2 Blinking LED from command line interface

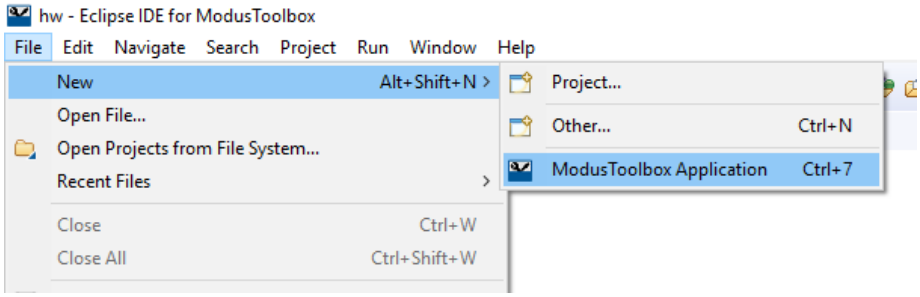
Repeat demonstration but using the command-line tools rather than an IDE.

5a.3 Exercises (Part 1)

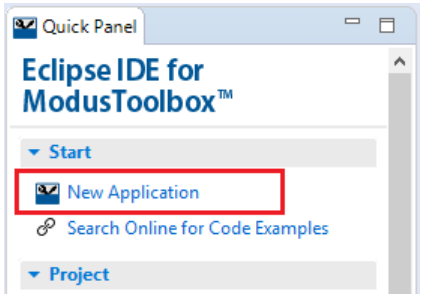
5a.3.1 Exercise 1: Blinking an LED from the Eclipse IDE

For the first example project, we will use the Eclipse IDE to create, build, and program the classic blinking LED project. I first created a new workspace called mtw101 under the “mtw” folder so that I can keep all the class exercises together, but you can call your workspace anything you like.

1. Select either **File > New > ModusToolbox Application**.

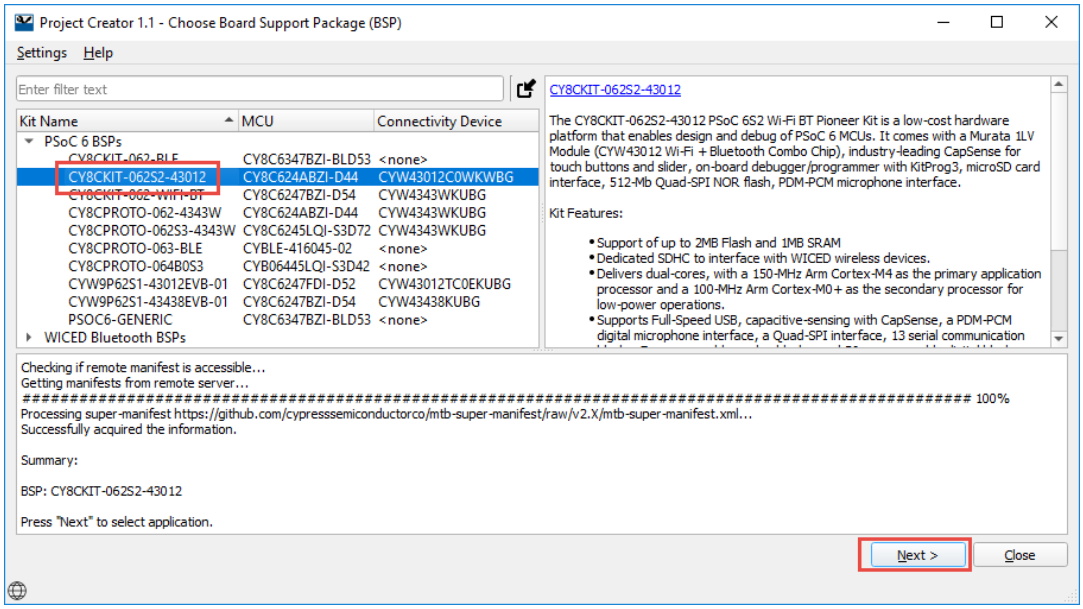


Or on the Quick Panel, click the **New Application** link.

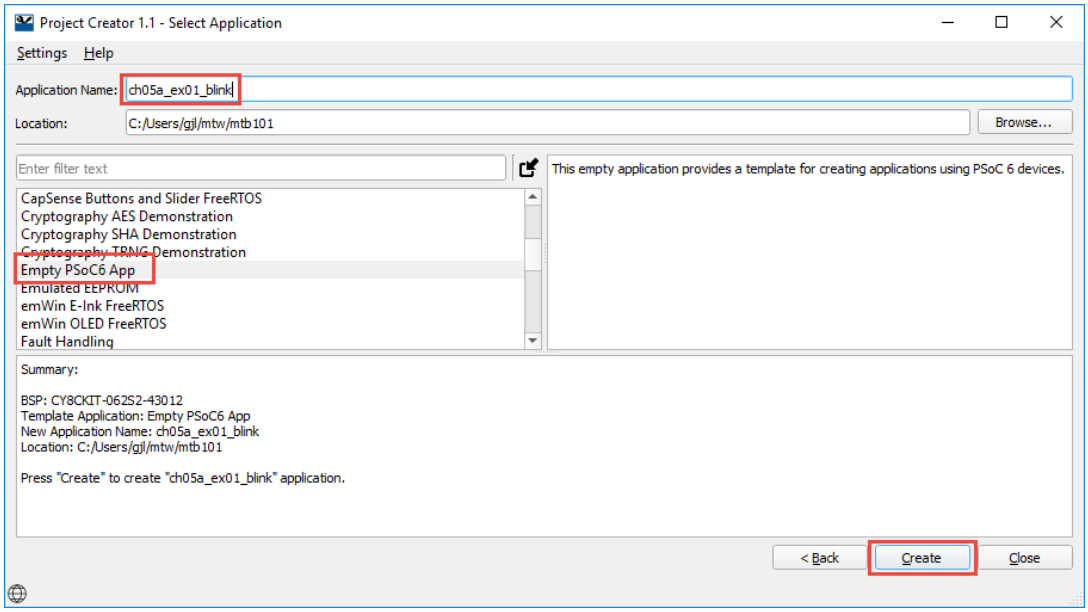


Either option launches the ModusToolbox Project Creator tool.

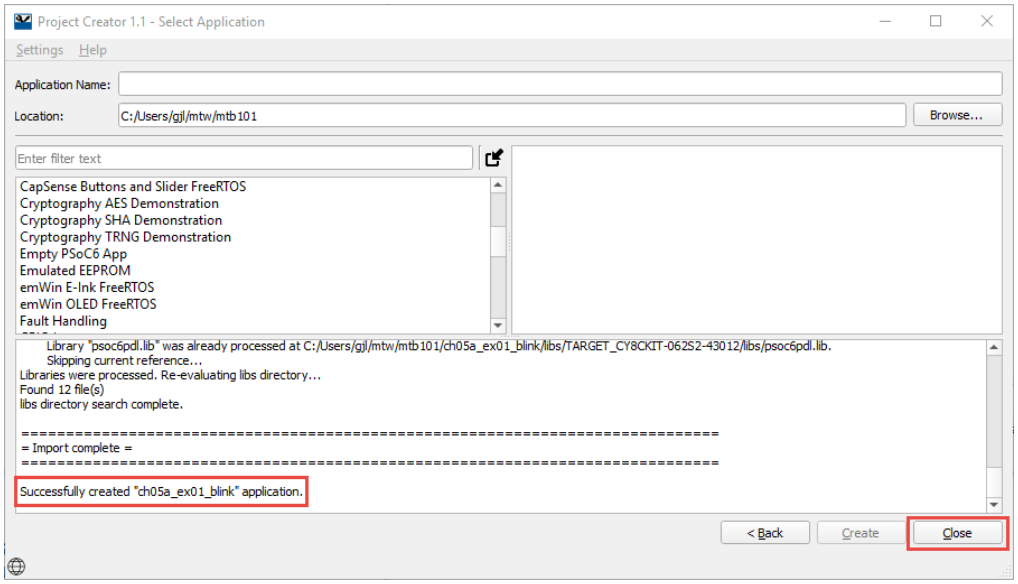
2. Choose the correct development kit and click **Next >**.



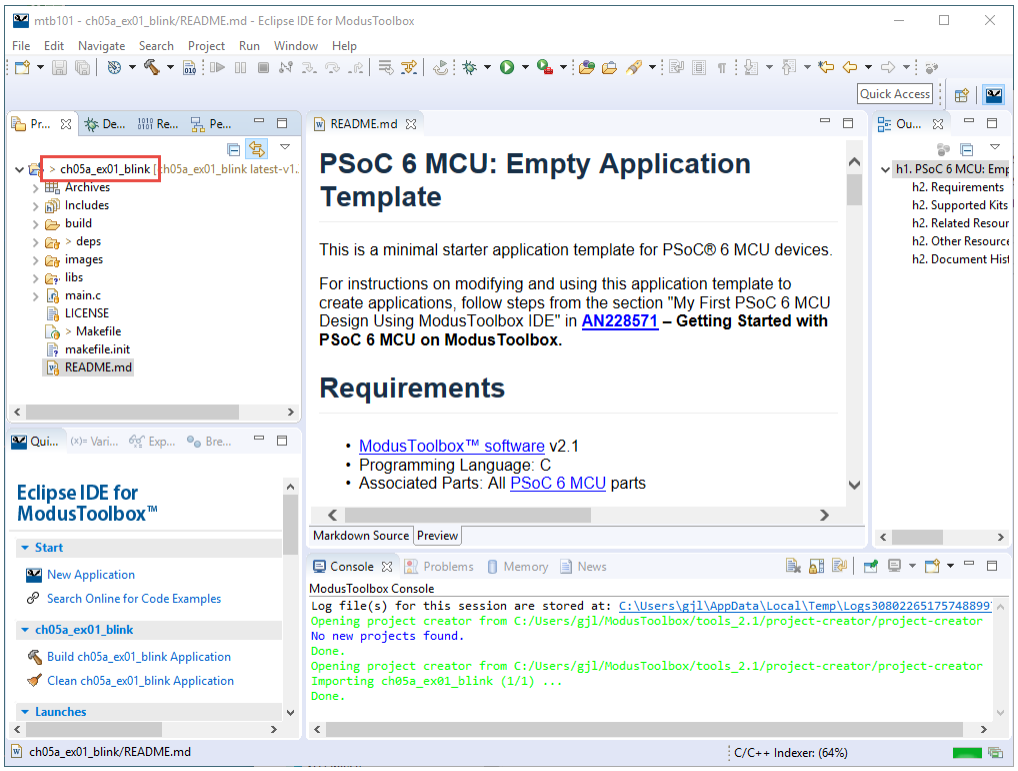
3. Select “Empty PSoC 6 App” and give your project a name (in this case **ch05a_ex01_blink**). Then, click **Create**.



4. After several moments, the application is created. Click **Close**.



Then, the application is imported into the Eclipse IDE.



5. Double click on main.c and change the code to look like this:

```
#include "cy_pdl.h"
#include "cyhal.h"
#include "cybsp.h"
int main(void)
{
    cy_rslt_t result;

    /* Initialize the device and board peripherals */
    result = cybsp_init() ;
    CY_ASSERT(result == CY_RSLT_SUCCESS);

    __enable_irq();

    cyhal_gpio_init(CYBSP_USER_LED1,CYHAL_GPIO_DIR_OUTPUT,
CYHAL_GPIO_DRIVE_STRONG, CYBSP_LED_STATE_OFF);

    for(;;)
    {
        cyhal_gpio_toggle(CYBSP_USER_LED1);
        Cy_SysLib_Delay(500);
    }
}
```

6. Finally, click on "ch05a_ex01_blink Program (KitProg3)" from the Quick Panel Launches.

You should now have a blinking LED. If you don't, then you probably need to switch the mode of the KitProg to be CMSIS-DAP Bulk or HID. Refer to the "Firmware Loader" section in Chapter 1 for details.

5a.3.2 Exercise 2: Blinking LED from the Command Line

You can build and program the same project using the command line. For more details, refer to section 5a.1.4 Command Line.

1. On Windows, navigate to the modus-shell directory and run Cygwin.bat.

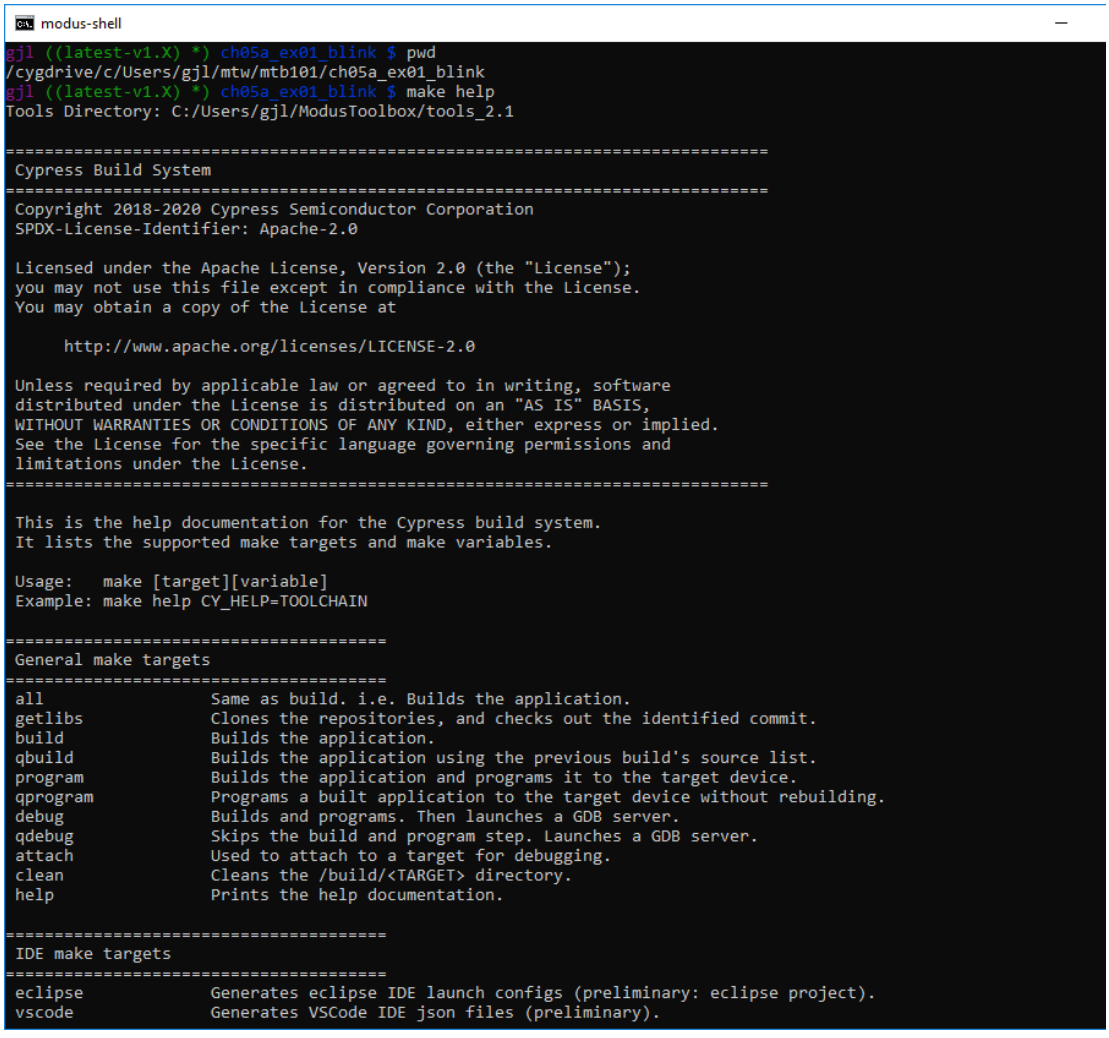
   <install_dir>\ModusToolbox\tools_2.1\modus-shell\

   On Linux or macOS, open a terminal window.

2. Navigate to the project directory that you created previously. For example:

   ```
   cd <user_home>/mtw/mtb101/ch05a_ex01_blink
   ```

3. Next, run `make help` to see the list of commands.



4. Then you can run:

   ```
   make clean
   make build
   make program
   ```

5a.4 Create ModusToolbox Projects

As we've discussed already, you can use the Eclipse IDE for ModusToolbox IDE to develop projects. We understand many developers may not want to use the Eclipse IDE. So, there are several ways to create projects (all based on the `git clone` mechanism). Here are some of the methods you can use:

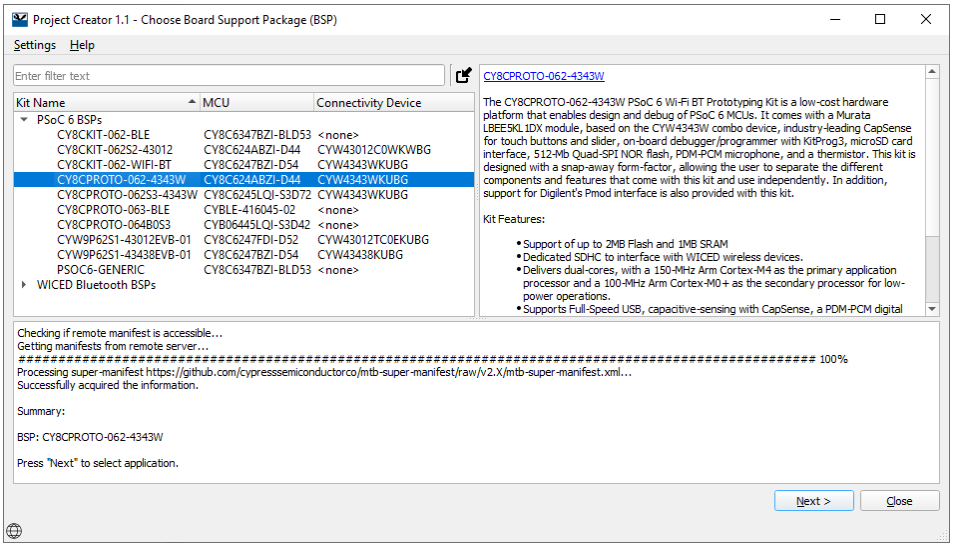
- Project Creator tool through Eclipse IDE (discussed previously)
- Stand-alone Project Creator tool
- Command Line / Manual Setup

You can also create projects in offline mode. See section 5a.12 for details.

5a.4.1 Project Creator Tool

Instead of using the Project Creator tool through the Eclipse IDE, you can create new projects with the tool in stand-alone mode. It is in the ModusToolbox/tools_2.1/project-creator directory. You can also just search for "project-creator".

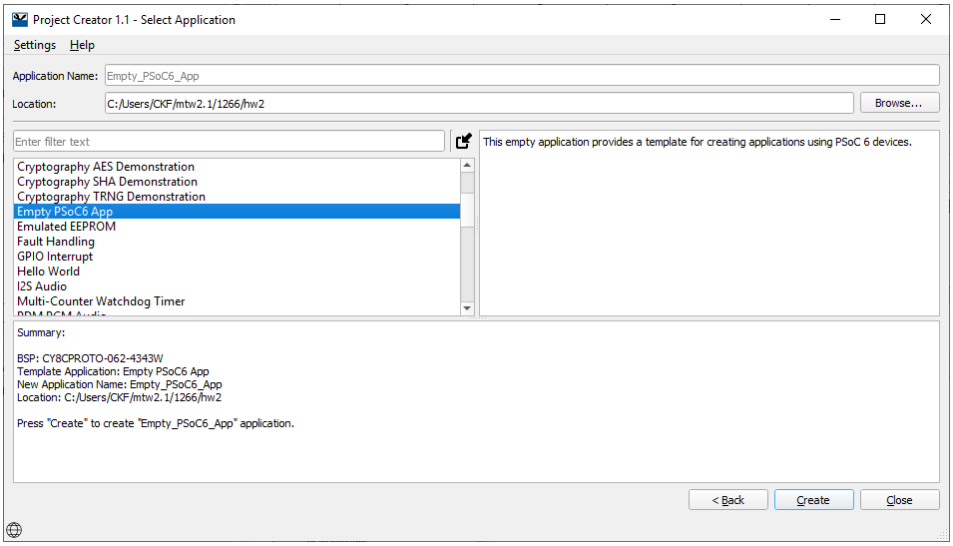
The first thing the project-creator tool does is read the Cypress configuration information from the Cypress GitHub site so that it knows all the BSPs etc. that we support.



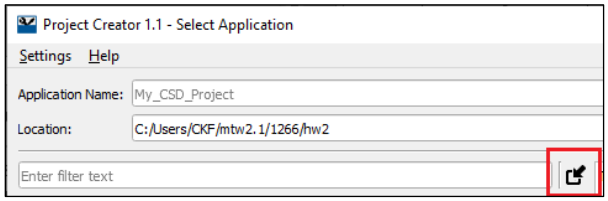
Note that there is also an import button (next to where you enter the kit name) so you can specify your own custom BSP if you have one. We will show you how to create your own BSP later in this chapter.

Select a kit and click **Next >**.

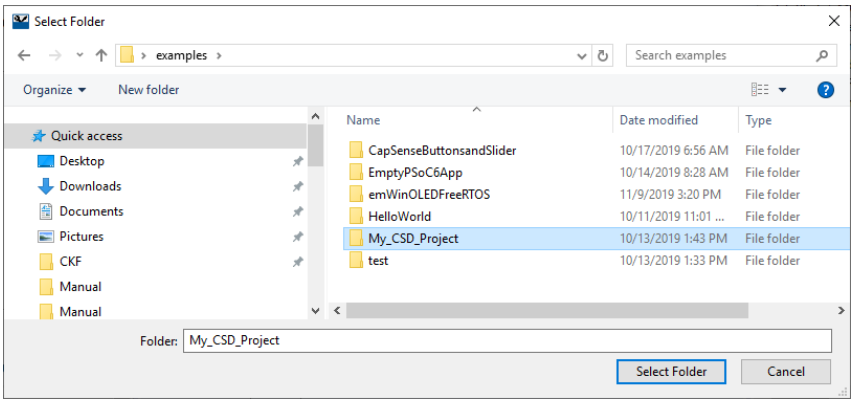
Now you will be presented with all the Cypress applications supported by the BSP you chose. Pick an application to start from, give it a name and specify a location for the project.



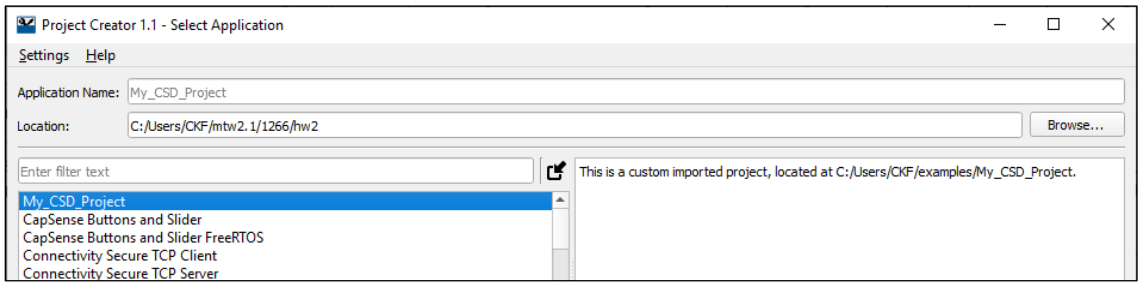
If you want to start with your own local project or template instead of one of the Cypress provided applications, click the **Import** button  .



This allows you to search your computer and then create a new application based on an old one. Make sure the path you select is to the directory that contains the Makefile (or one above it – more on that in a minute) – otherwise the import will fail.



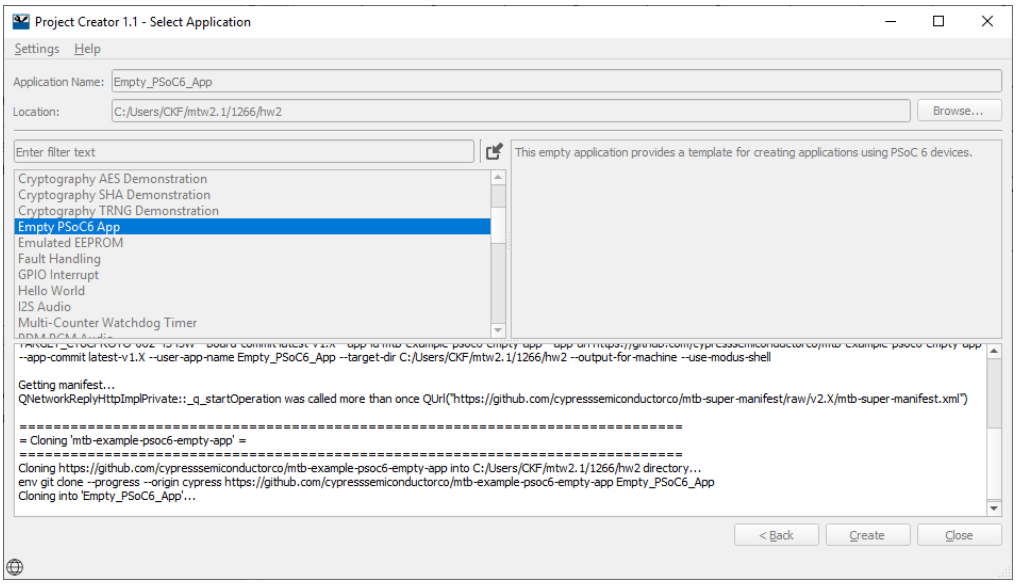
Once you select an application folder, it will show up at the top of the list of applications.



Note that the existing project can be one that is in your workspace or one that is located somewhere else. It does not need to be a full Eclipse project. At a minimum, it needs a Makefile and source code files, but it may contain other items such as configurator files and .lib files.

If you specify a path to a directory that is above the Makefile directory, the hierarchy will be maintained, and the path will be added to each individual application name. This can be useful if you have a directory containing multiple applications in sub-directories and you want to import them all at once. For example, if you have a directory called “myapps” containing the 2 subdirectories “myapp1” and “myapp2”, when you import from the “myapps” level you will get a project called myapps.myapp1 and myapps.myapp2.

Once you have selected the application you want to create (whether it's a Cypress provided application or one of your own), click **Create** and the tool will create the application for you.



The tool runs the `git clone` operation and then the `make getlibs` operation. The project is saved in the directory you specified previously. When finished, click **Close**.

Note that you can create more than one project before clicking close if desired. Just select each project and click create one at a time until you have created all the applications that you want.

5a.4.2 Command Line / Manual Setup

Project Creator CLI

The Project Creator tool described above also has a command line version that you can run from a shell (e.g. Cygwin or modus-shell). It is in the same folder as the Project Creator GUI (ModusToolbox/tools_2.1/project-creator) but the executable is called project-creator-cli.exe. You can run it with no arguments to see the different options available. You can run it to see a listing of all the available BSPs:

```
./project-creator-cli --list-boards
```

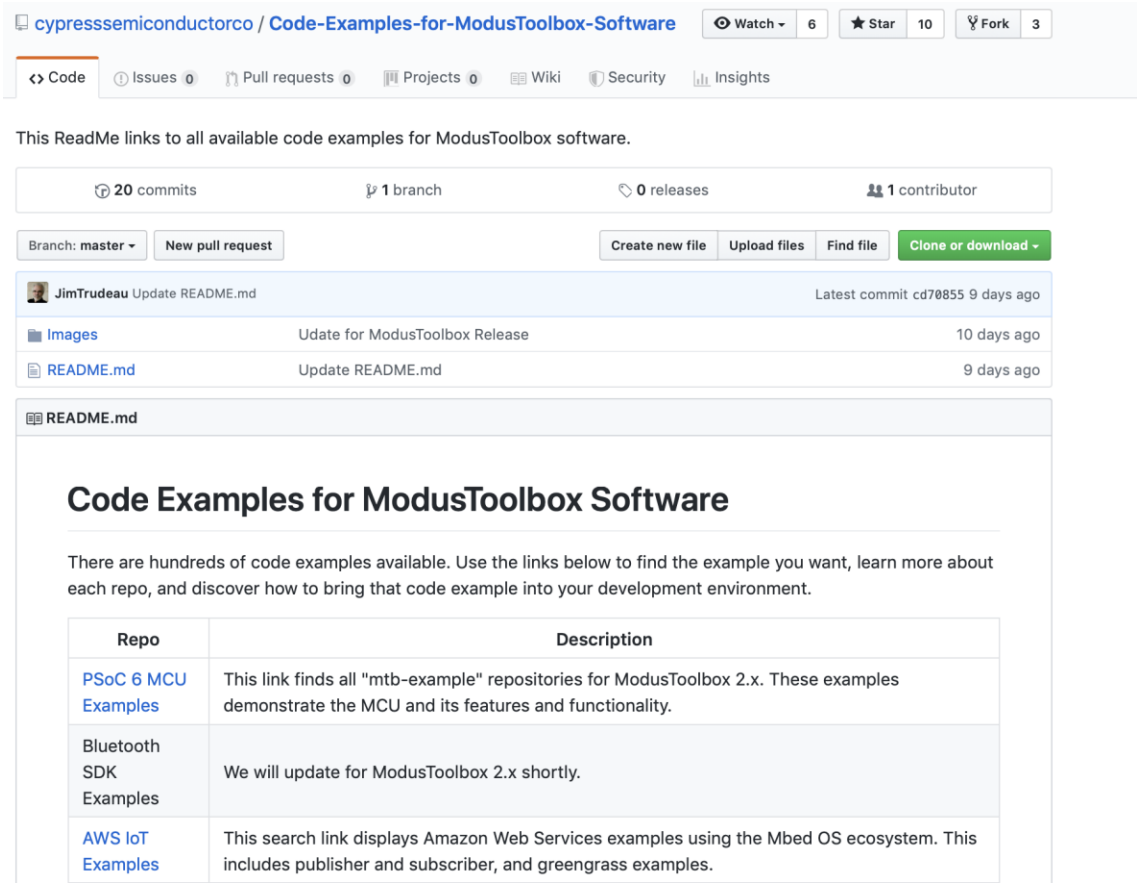
You can also see all the available applications for a given BSP:

```
./project-creator-cli --list-apps <BSP_Name>
```

Using GitHub

Instead of using the Project Creator CLI tool, you can use git commands directly to clone a project.

You can get to the Cypress GitHub repo using the “Search Online for Code Examples” link in the Eclipse IDE Quick Panel. That will take you the following website. You can use the GitHub website tools to clone projects. You can also use the command line to clone as described in the next section.



Using Git from the Command Line

If you know the URL and don't want to use GitHub, from a shell you can use the `git clone` command to clone the project:

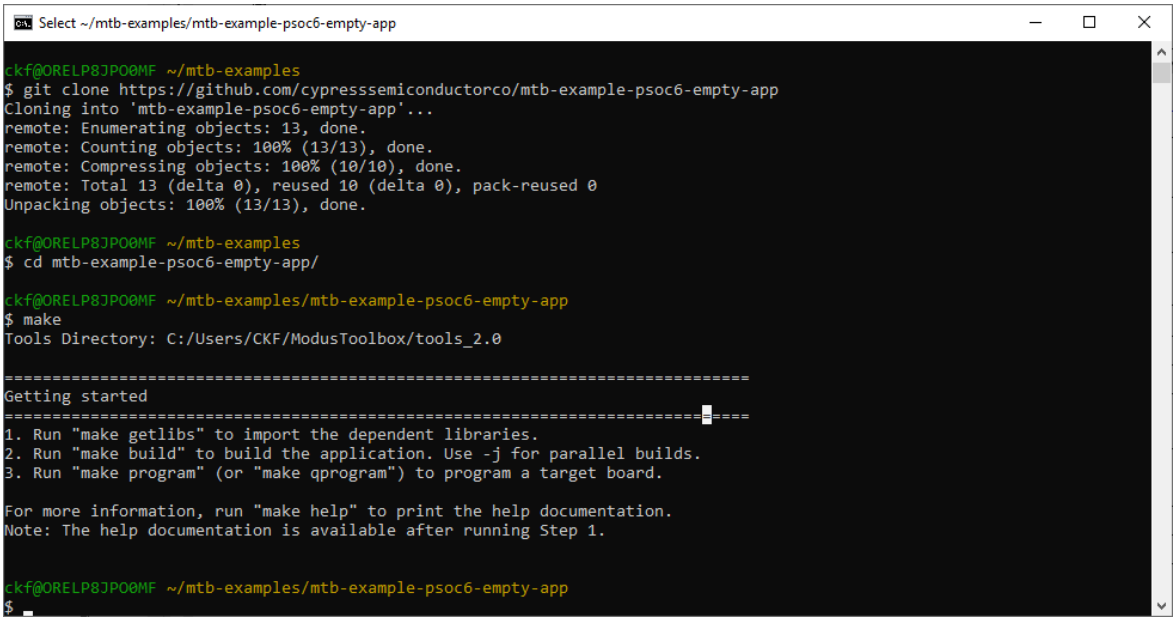
```
git clone https://github.com/cypresssemiconductorco/mtb-example-psoc6-empty-app
```

Make

Once you have cloned the project from GitHub or the command line, change to the project's directory:

```
cd mtb-example-psoc6-empty-app
```

Run `make` to see the list of available commands:



Then you can run:

```
make getlibs
make build
make program
```

Note: If you used the CY8CKIT-062S2-43012 kit, programming will not succeed since the default library and TARGET are set for the CY8CPROTO-062-4343W kit. You will need to manually add the CY8CKIT-062S2-43012 library to the project (or use the Library Manager to add it) and change the TARGET in the Makefile (or specify it on the command line). See section 5a.7 Libraries & Library Manager for more details about libraries.

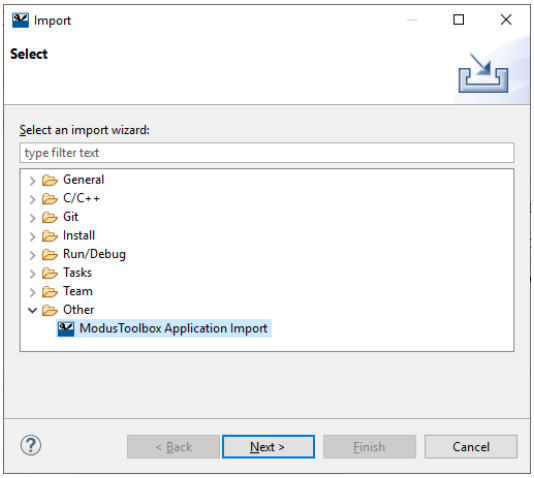
5a.5 Importing/Exporting Applications

Once you have an application created with the Project Creator tool, you can use it from the command line, or you can convert it to use in the IDE of your choosing. You can even switch back and forth between the command line and IDE.

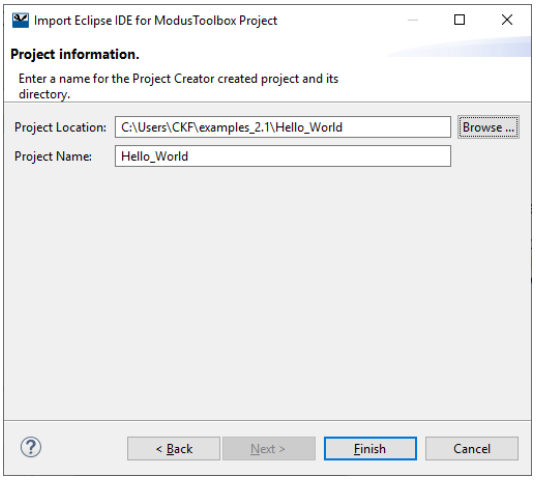
To open a command-line application in an IDE, you need to run a `make <IDE>` command to generate appropriate files. When you create an application starting from the Eclipse IDE, the `make eclipse` command is run for you automatically. There is also an import option in the Eclipse IDE for ModusToolbox that runs `make eclipse` plus a few other required actions, so it isn't necessary for you to run `make eclipse` first. For other IDEs (VS Code, IAR Embedded Workbench, and Keil µVision), you must run the appropriate `make <IDE>` command manually prior to opening the application in the IDE.

5a.5.1 Import Projects into the Eclipse IDE

If you have a project that was created from the command line or stand-alone Project Creator tool, you can use the Eclipse Import feature: **File > Import… > Other > ModusToolbox Application Import**. Note that this imports the application in-place; it does NOT copy it into your workspace. If you want the resulting application to be in your workspace, make sure you put it in the workspace folder before importing it.



After clicking **Next >**, browse to the location of the application's directory, select it, and click **Finish**.



This import mechanism runs `make eclipse` plus various commands to help the application run smoothly in the Eclipse IDE for ModusToolbox.

Launch Configs

A few important notes regarding the launch configurations inside Eclipse:

There is a directory inside the application called *.mtbLaunchConfigs*. This is where the launch configurations are located for an application. There are cases where you will end up with old launch configurations in that directory which can confuse Eclipse. You may see no launch configs, the wrong launch configs, or multiple sets of launch configs.

In case you see that behavior, it is safe to blow away the *.mtbLanuncConfigs* directory at any time and then just click on "Generate Launches for <project name>" in the Quick Panel.

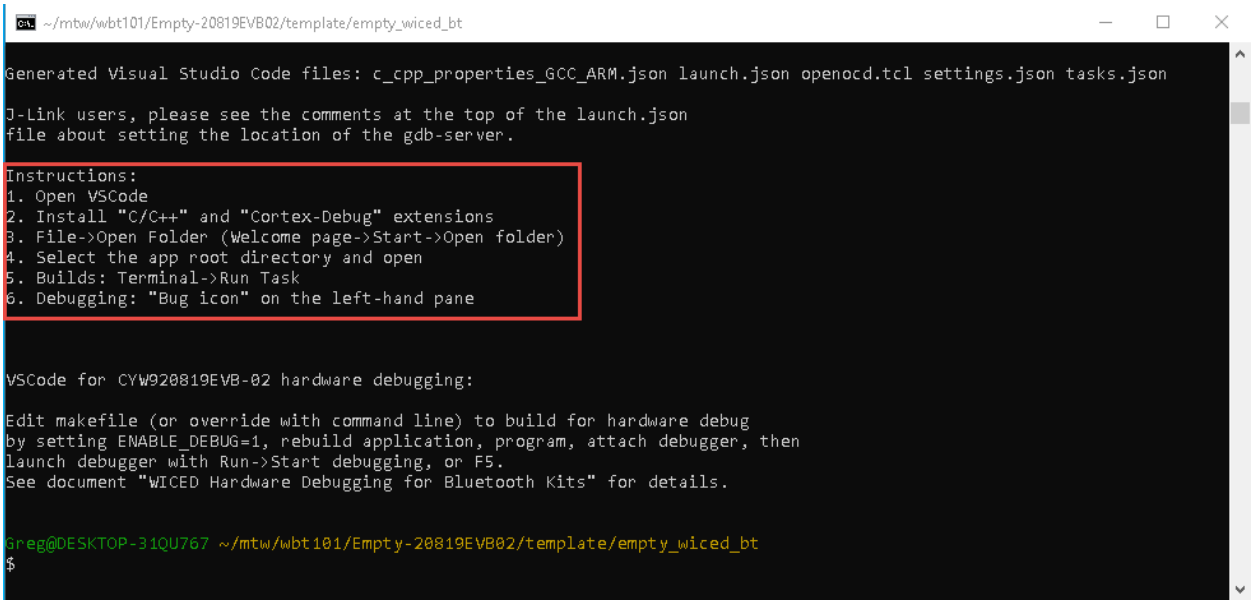
Some of the cases where you may end up with stale or multiple sets of launch configs:

- If you rename an application directory before importing.
- If you rename an application inside of Eclipse (that is, **right-click > Rename**).
- If you create a new ModusToolbox application and point to an existing application as the starter template.

5a.5.2 Export for Visual Studio (VS) Code

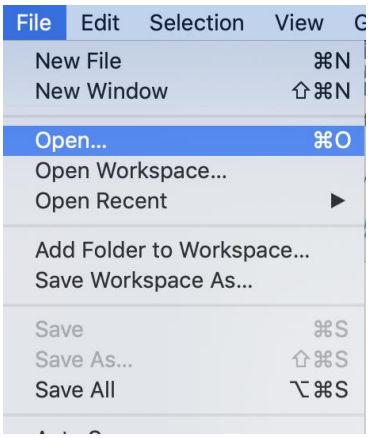
One alternative to using the Eclipse-based IDE is a code editor program called VS Code. This tool is quickly becoming a favorite editor for developers. The ModusToolbox command line knows how to make all the files required for VS Code to edit, build, and program a ModusToolbox program. To create these files, go to an existing project directory and type the following from the command line:

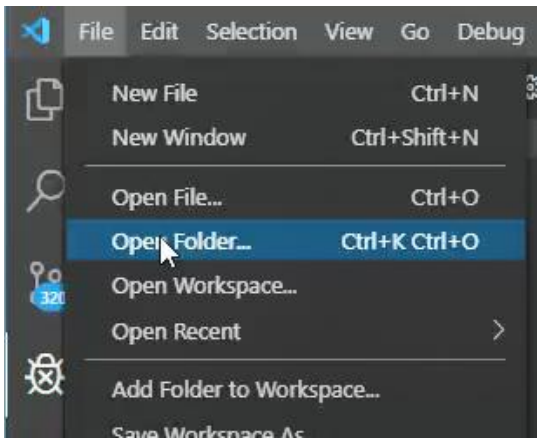
```
make vscode
```



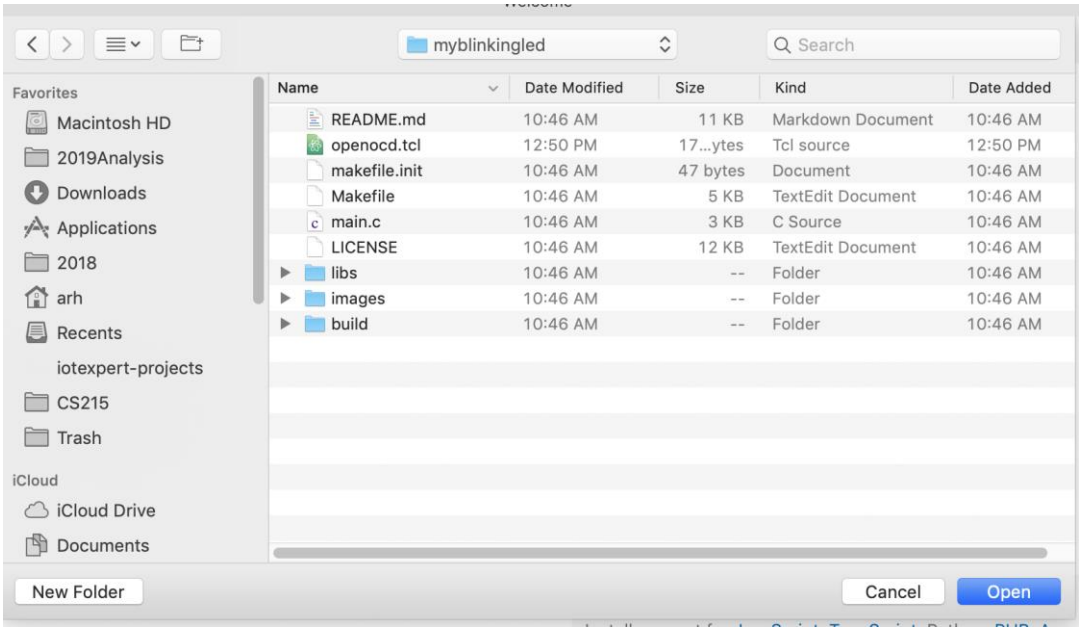
The message at the bottom of the command window tells you the next steps to take. These steps are explained as follows:

1. Start VS Code.
2. Install the C/C++ and Cortex-Debug extensions if you don't already have them.
3. Open your application in Visual Studio Code using **File > Open** (on macOS) or **File > Open Folder** (on Windows).



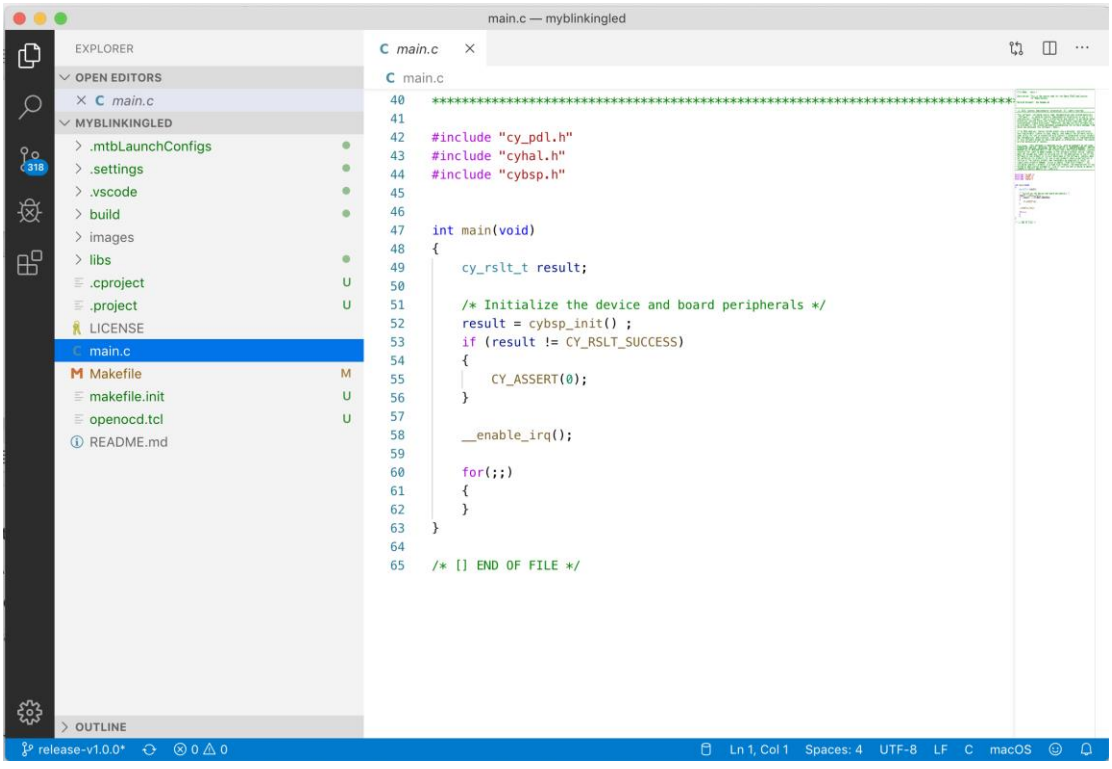


4. Then, navigate to the directory where your project resides and click **Open**.

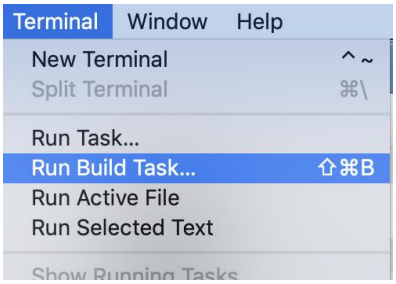


Hint: Alternately, you can just enter `code .` on the command line after running `make vscode` and it will open VS Code and load the application all in one step.

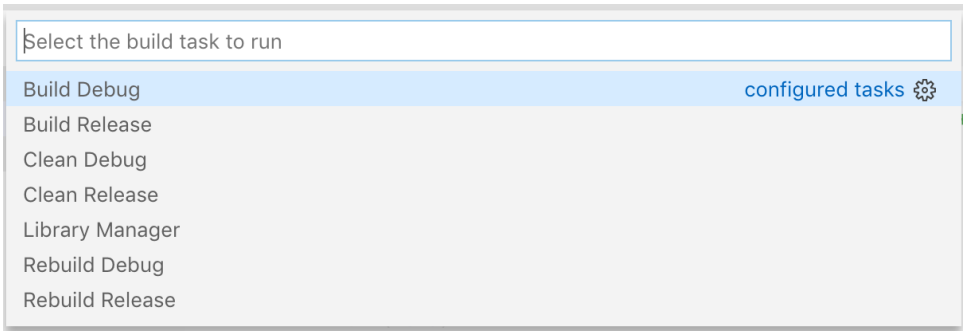
Now your windows should look something like this. You can open the files by clicking on them in the Explorer window.



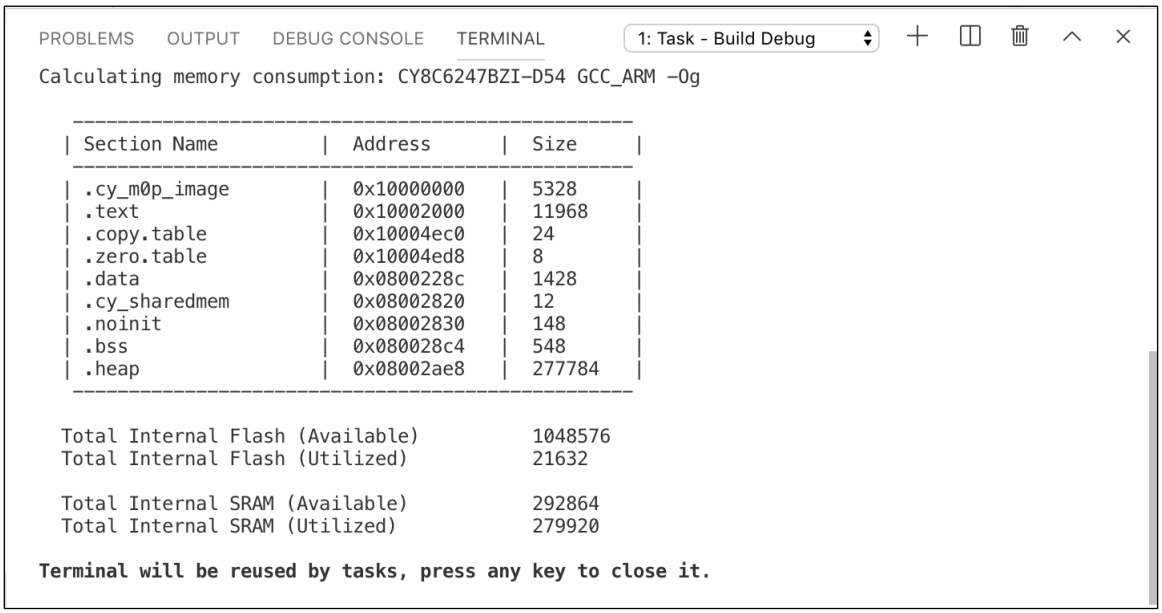
5. In order to build your project, select **Terminal > Run Build Task…**



Then, select **Build Debug**. This will essentially run `make build` at the top level of your project.



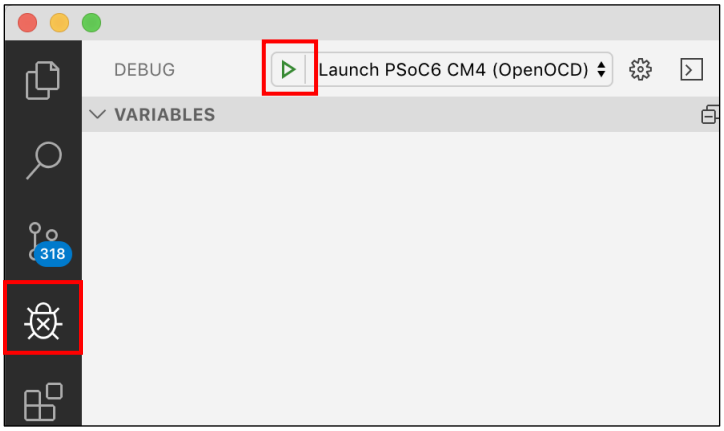
You should see the normal compiler output in the console at the bottom of VS Code:



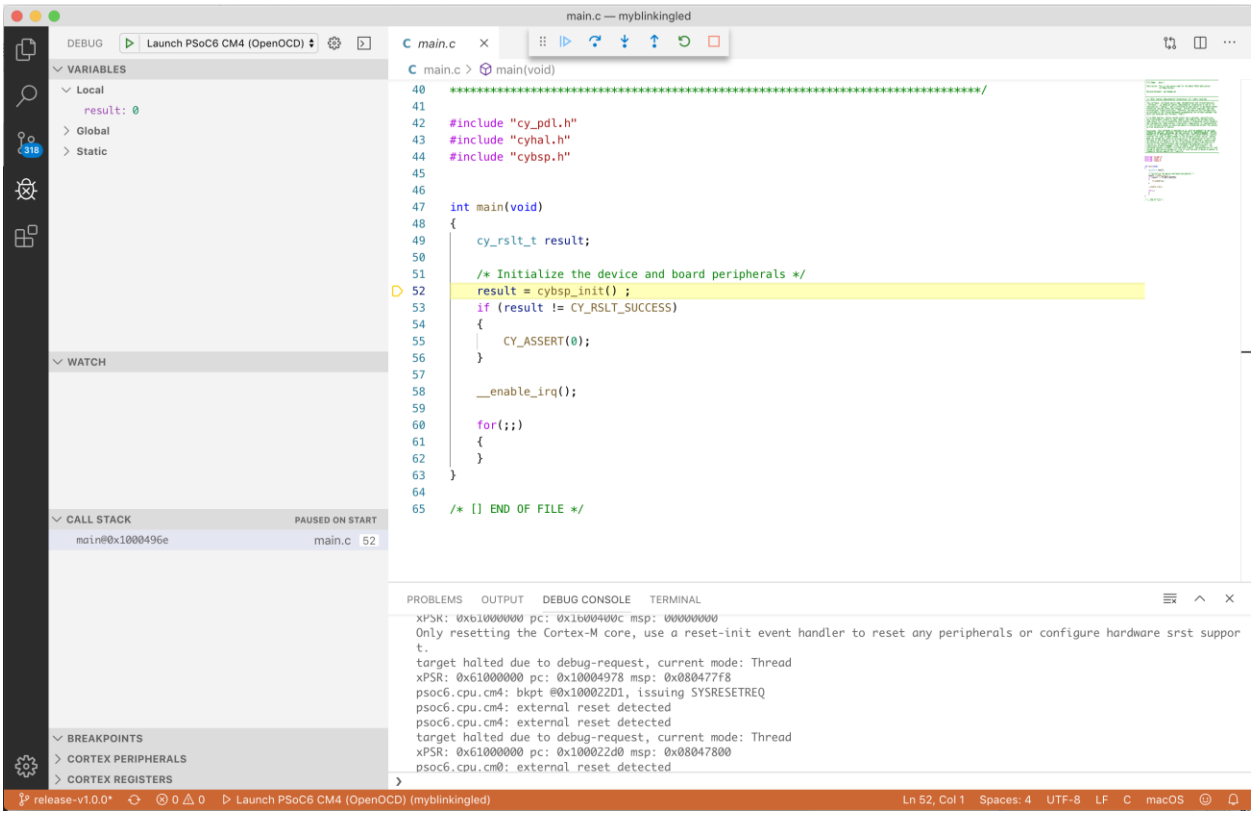
6. To program the kit or run the debugger, first make sure you have installed the “Cortex-Debug” VS Code plug in from the marketplace. See the software installation instructions if you didn't do this when you installed VS Code.

   Typically, you will need to build first, then program or program and start the debugger. The program and debug can be done in one step but not the build. That is, you don't need to explicitly program before debugging, but you do need to build first to see any changes.

   To do programming or debug, click the little “bug” icon on the left side of the screen. Then click the drop-down next to the green **Play** button to select Program (to program), Launch PSoC 6 CM4 (to program and debug), or Attach PSoC 6 CM4 (to debug without programming).



If you chose one of the debugging options, after clicking **Play**, your screen should look something like this. Your application has been programmed into the chip (if you chose the Launch option). The reset of the chip has happened, and the project has run until "main". Notice the yellow highlighted line. It is waiting on you. You can now add breakpoints or get the program running or single step.



5a.5.3 Export for IAR Embedded Workbench

The ModusToolbox build system also provides a make target for IAR Embedded Workbench. After creating a ModusToolbox application, run the following command:

```
make ewarm8
```

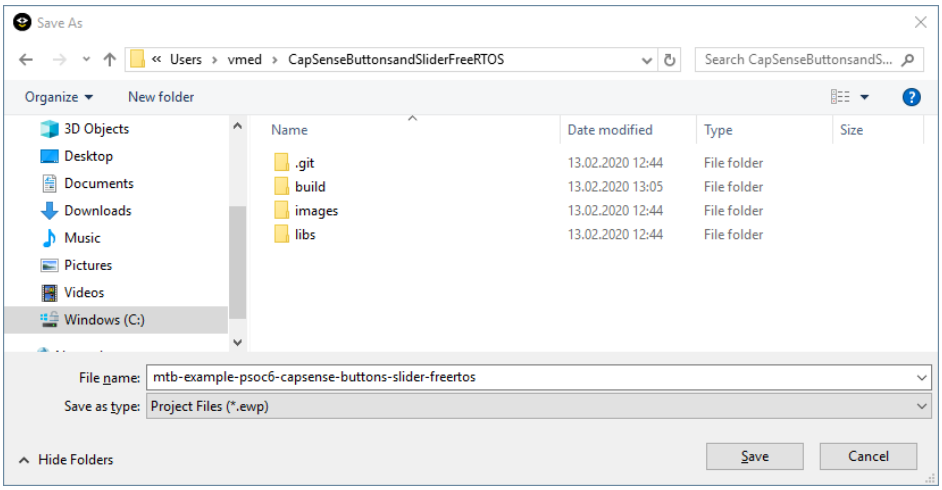
Note that export to IAR Embedded Workbench is **not** currently supported for BT SoC solution applications.

An IAR connection file appears in the application directory:

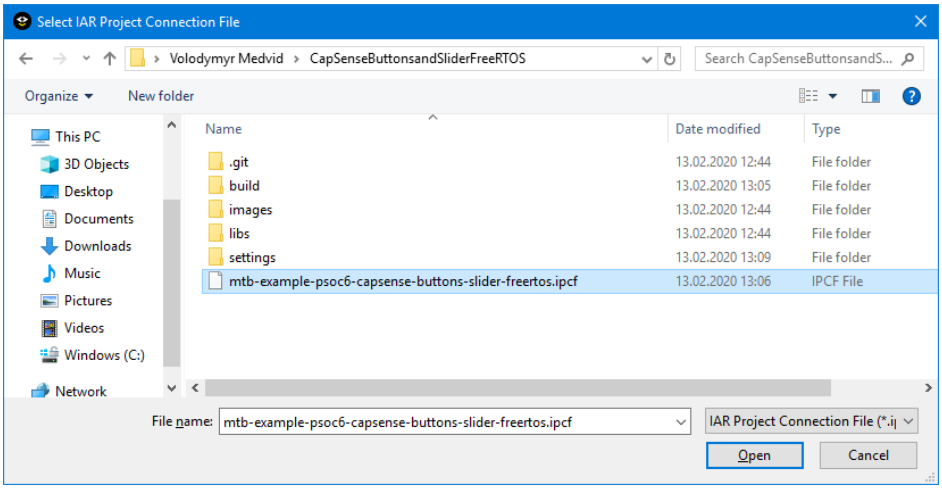
- <project-name>.ipcf

1. Start IAR Embedded Workbench.
2. On the main menu, select **Project > Create New Project > Empty project** and click **OK**.

3. Browse to the ModusToolbox application directory, enter an arbitrary application name, and click **Save**.



4. After the application is created, select **File > Save Workspace**, enter an arbitrary workspace name and click **Save**.
5. Select **Project > Add Project Connection** and click **OK**.
6. On the Select IAR Project Connection File dialog, select the *.ipcf* file and click **Open**:



7. On the main menu, Select **Project > Make**.

At this point, the application is open in the IAR Embedded Workbench. There are several configuration options in IAR that we won't discuss here. For more details about using IAR, refer to the "Exporting to IDES" chapter in the ModusToolbox User Guide.

5a.5.4 Export for Keil µVision

The ModusToolbox build system also provides a make target for Keil µVision. After creating a ModusToolbox application, run the following command:

```
make uvision5
```

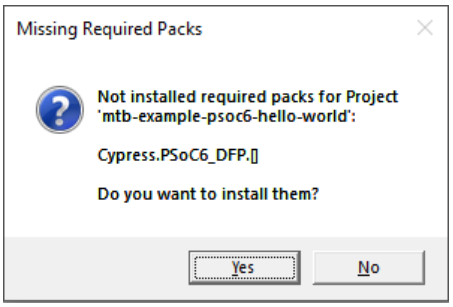
Note that export to Keil µVision is **not** currently supported for BT SoC solution applications.

This generates two files in the application directory:

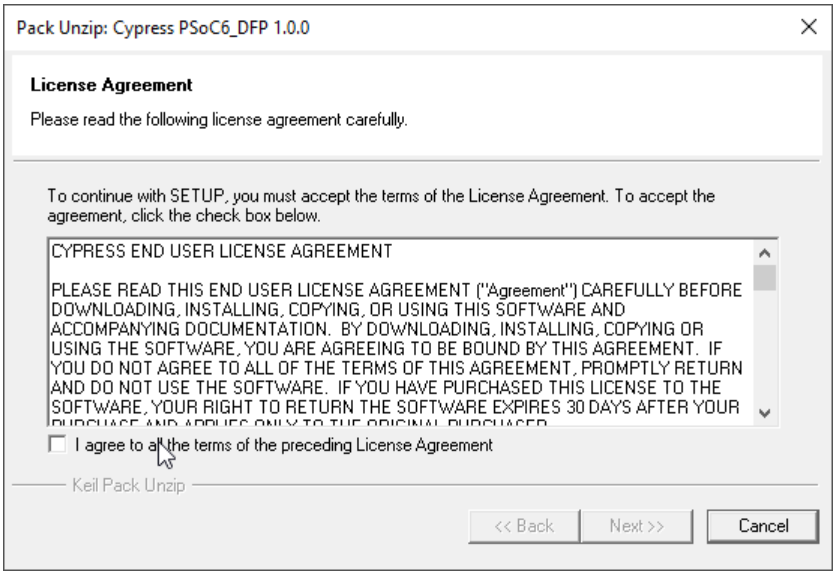
- <project-name>.cpdsc
- <project-name>.gpdsc

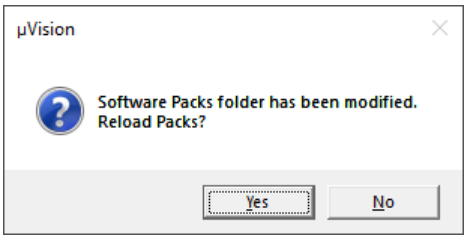
The cpdsc file extension should have the association enabled to open it in Keil µVision.

1. Double-click the *mtb-example-psoc6-hello-world.cpdsc* file. This launches Keil µVision IDE. The first time you do this, the following dialog displays:



2. Click **Yes** to install the device pack. You only need to do this once.
3. Follow the steps in the Pack Installer to properly install the device pack.



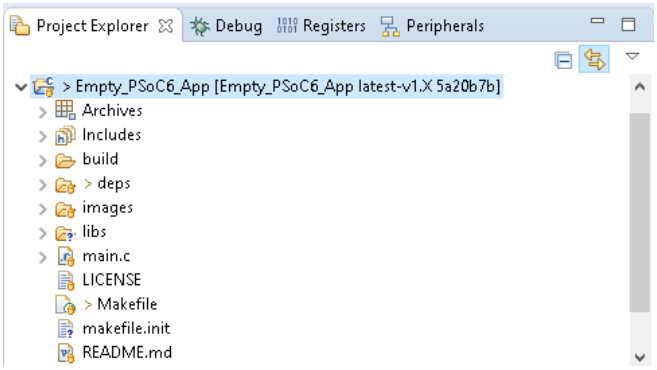


4. When complete, close the Pack Installer and close the Keil µVision IDE.
5. Then double-click the .cpdsc file again and the application will be created for you in the IDE.

At this point, the application is open in the Keil µVision IDE. There are several configuration options in µVision that we won't discuss here. For more details about using µVision, refer to the "Exporting to IDES" chapter in the ModusToolbox User Guide.

5a.6 Project Directory Organization

Once created, a typical ModusToolbox PSoC 6 project contains various files and directories.



5a.6.1 Archives

Virtual directory that shows static libraries.

5a.6.2 Includes

Virtual directory that shows the include paths used to find header files.

5a.6.3 build

This directory contains build files, such as the .elf and .hex files.

5a.6.4 deps

This directory contains .lib files that specify where the `make getlibs` command finds the libraries used by the project. Note that in some applications, the .lib files may be in the libs directory along with the libraries themselves.

5a.6.5 images

This directory contains artwork images used by the documentation.

5a.6.6 libs

The libs directory contains all the code libraries and supporting files, including the targets, BSP, HAL, PDL, firmware, components, and documentation. These are covered in later sections.

5a.6.7 main.c

The is the primary application file that contains the project's application code.

5a.6.8 LICENSE

This is the ModusToolbox license agreement file.

5a.6.9 Makefile

The Makefile is used in the application creation process. It defines everything that ModusToolbox needs to know to create/build/program the application. This file is interchangeable between Eclipse IDE and the Command Line Interface so once you create an application, you can go back and forth between the IDE and CLI at will.

Various build settings can be set in the Makefile to change the build system behavior. These can be "make" settings or they can be settings that are passed to the compiler. Some examples are:

Target Device (`TARGET=`)

```
27  ################################################################################
28  # Basic Configuration
29  ################################################################################
30
31  # Target board/hardware
32  TARGET=CY8CKIT-062-WIFI-BT
```

Build Configuration (`CONFIG=`)

```
47  # Debug   -- build with minimal optimizations, focus on debugging.
48  # Release -- build with full optimizations
49  CONFIG=Debug
```

Adding Components (`COMPONENTS=`)

```
################################################################################
# Advanced Configuration
################################################################################

# Enable optional code that is ordinarily disabled by default.
#
# Available components depend on the specific targeted hardware and firmware
# in use. In general, if you have
#
#    COMPONENTS=foo bar
#
# ... then code in directories named COMPONENT_foo and COMPONENT_bar will be
# added to the build
#
COMPONENTS=

# Like COMPONENTS, but disable optional code that was enabled by default.
DISABLE_COMPONENTS=
```

5a.6.10 Makefile.init

This file contains variables used by the make process. This is a legacy file not needed any more.

5a.6.11 README.md

Almost every code example / starter application includes a README.md (mark down) file that provides a high-level description of the project.

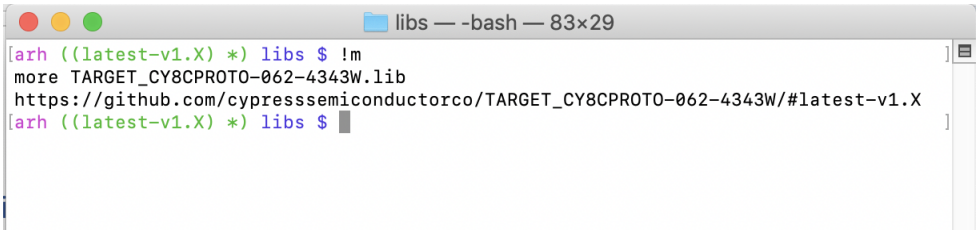
5a.7 Libraries & Library Manager

A ModusToolbox project is made up of your application files plus libraries. A library is a related set of code, either in the form of C-source or compiled into archive files. These libraries contain code which is used by your application to get things done. These libraries range from low-level drivers required to boot the chip, to the configuration of your development kit (called Board Support Package) to Graphics or RTOS or CapSense, etc.

ModusToolbox knows about a library in your project by having a “dot lib” file somewhere in the directories of your project (typically in the deps or libs directory but can be anywhere). A “dot lib” file is simply a text file that has two parts:

- A URL to a Git repository somewhere that is accessible by your computer such as GitHub
- A Git Commit Hash or Tag that tells which version of the library that you want

A typical library file will look something like this (notice this uses the tag name for the version).

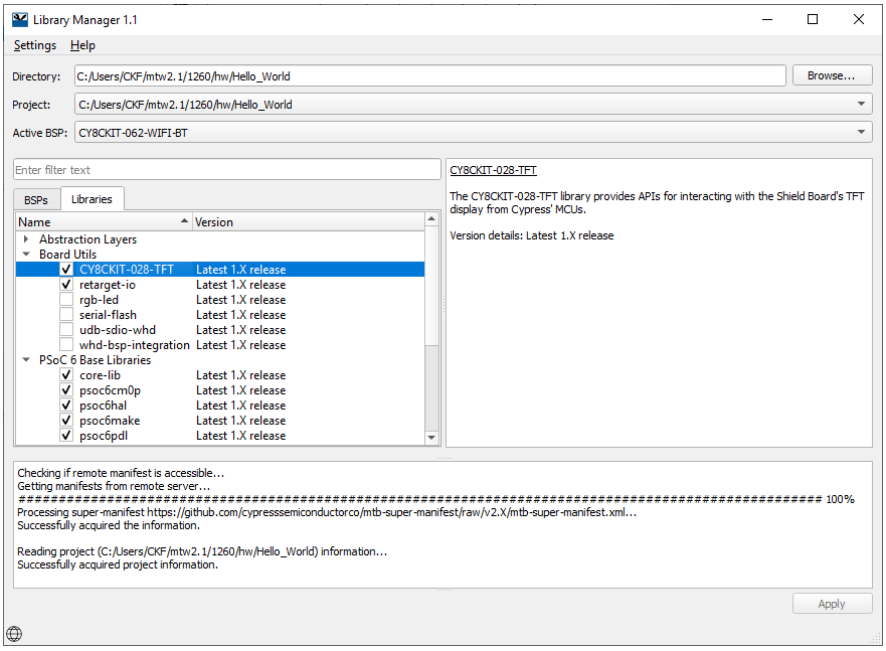


In a typical application, there will be a directory called “deps” which will contain the .lib files needed for the application. In order to actually get the library files into the application, you need to run the command line function `make getlibs` (or run the library manager). This will search your project, find all the “dot lib” files, and bring in the libraries to your project. The libraries will go in a directory called libs. Since the libraries are all pulled in using `make getlibs`, you don't typically need to check them in to a revision control system - they can be recreated at any time from the .lib files by re-running `make getlibs`.

The default version is a tag with the word “latest” that will get the latest version of the library, but you can enter a specific tag or commit to get a particular version of a library. This is important when you have a design going into production and want to guarantee that nothing changes out from under you.

5a.7.1 Library Manager

ModusToolbox provides a GUI for helping you manage library files. You can run this from the Quick panel link “Library Manager”. It is also available outside the IDE from ModusToolbox/tools_2.1/library-manager.



The Library Manager provides a GUI to select which Board Support Package (BSP) and version should be used by default when building a ModusToolbox application. An application can contain multiple BSPs, but a build from the IDE will build and program for the "Active BSP" selected in the library manager. The tool also allows you to add and remove libraries, as well as change their versions.

Note: The column above that says "Version". This allows you to lock to specific library versions right from the tool instead of having to manually edit .lib files.

When you run the Library Manager GUI, it will create ".lib" files. Then it will run `make getlibs`. You can always do this for yourself.

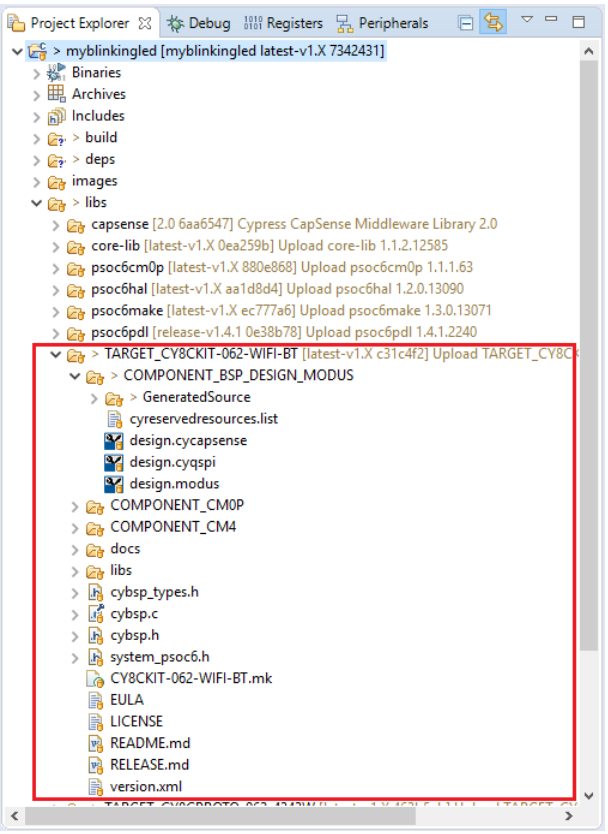
Libraries can be hierarchical, meaning that a library can have ".lib" files inside of it and the `make getlibs` command will bring in all the libraries through the hierarchy.

The Library Manager knows where to find libraries by using Manifest files, which will be discussed in the Manifests section. Therefore, .lib files included in your application that are not in the Manifest file will NOT show up in the Library Manager. This may include your own custom libraries or libraries that you got from a source other than Cypress. Since they don't show up in the Library Manager, they need to be added/removed/edited manually. However, `make getlibs` will still work on those .lib files as long as they point to a valid git repository.

5a.8 Board Support Packages

Each project is based on a target set of hardware. This target is called a "Board Support Package" (BSP). It contains information about the chip(s) on the board, how they are programmed, how they are connected, what peripherals are on the board, how the device pins are connected, etc. A BSP directory starts with the keyword "TARGET" and can be seen in the project directory as shown here:

5a.8.1 BSP Directory Structure



deps

For newer ModusToolbox applications, this directory contains.lib files that specify the required libraries for the underlying device family. These are provided by Cypress and always point to a Cypress owned repo or a repo that is sanctioned by Cypress. You can use the `make getlibs` command to fetch these libraries. The Project Creator and Library Manager tools invoke `make getlibs` automatically.

libs

This directory contains all the libraries after the `make getlibs` command has been invoked.

This directory stores the imported library files after running `make getlibs` to process the .lib files. For older versions of ModusToolbox applications, this directory also contains .lib files. The build system supports both.

COMPONENT_BSP_DESIGN_MODUS

This directory contains the configuration files (such as design.modus) for use with various BSP configurator tools, including the Device Configurator, QSPI Configurator, and CapSense Configurator. At the start of a build, the build system invokes these tools to generate the source files in the GeneratedSource directory.

COMPONENT_CM4 and COMPONENT_CM0P

These directories contain startup code and linker scripts for all supported toolchains for each of the two cores - the CM4 and the CM0+.

docs

The docs directory contains the HTML based documentation for the BSP.

cybsp_types.h

The cybsp_types.h file contains the aliases (macro definitions) for all the board resources.

cybsp.h / cybsp.c

These files contain the API interface to the board's resources.

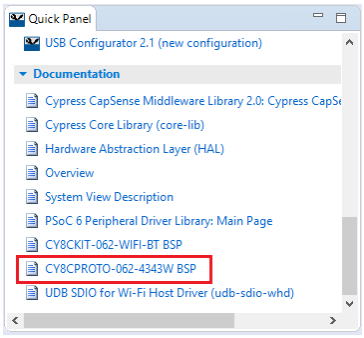
You need to include only cybsp.h in your application to use all the features of a BSP. Call the cybsp_init() function from your code to initialize the board (that function is defined in cybsp.c).

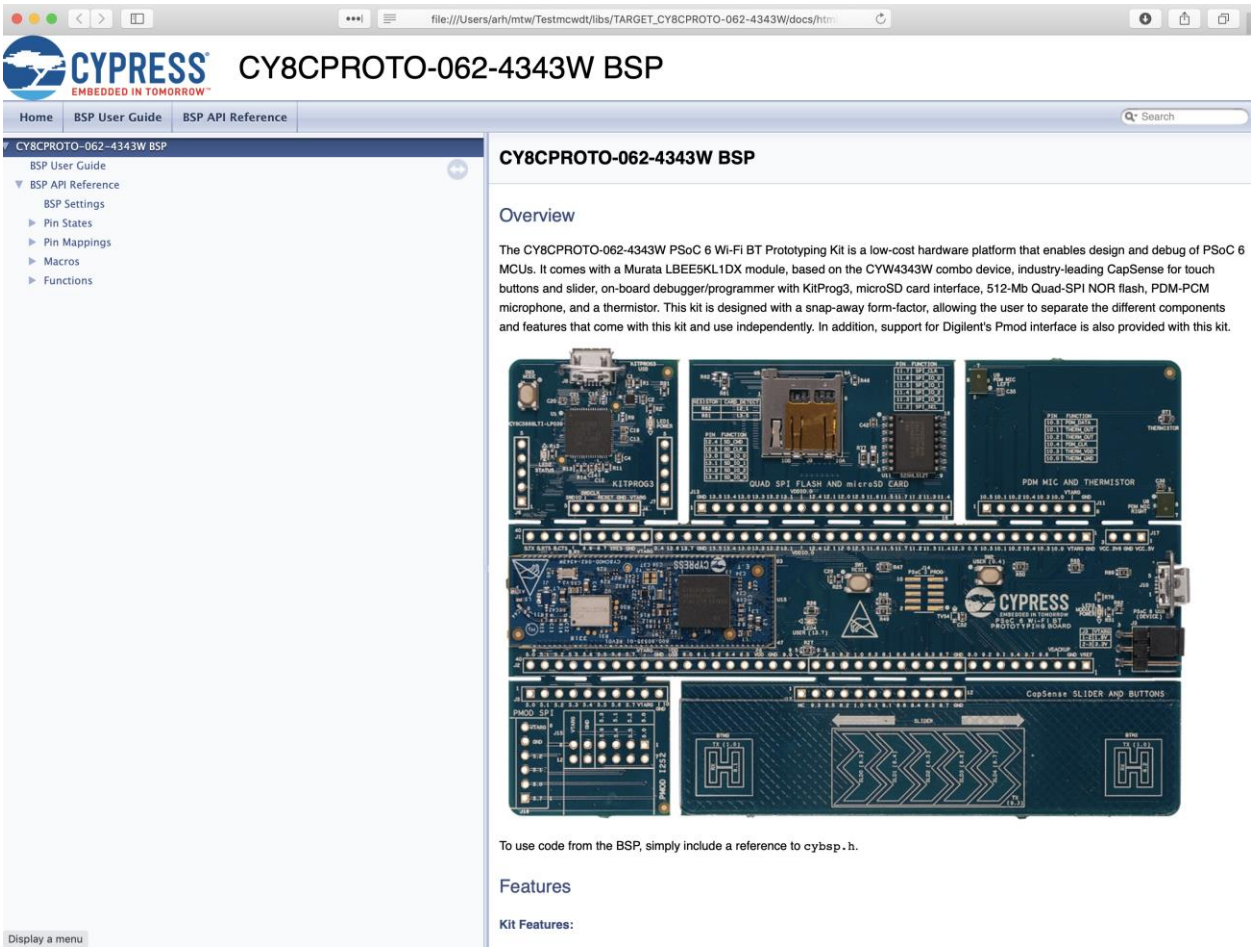
<targetname>.mk

This file defines the DEVICE and other BSP-specific make variables such as COMPONENTS.

5a.8.2 BSP Documentation

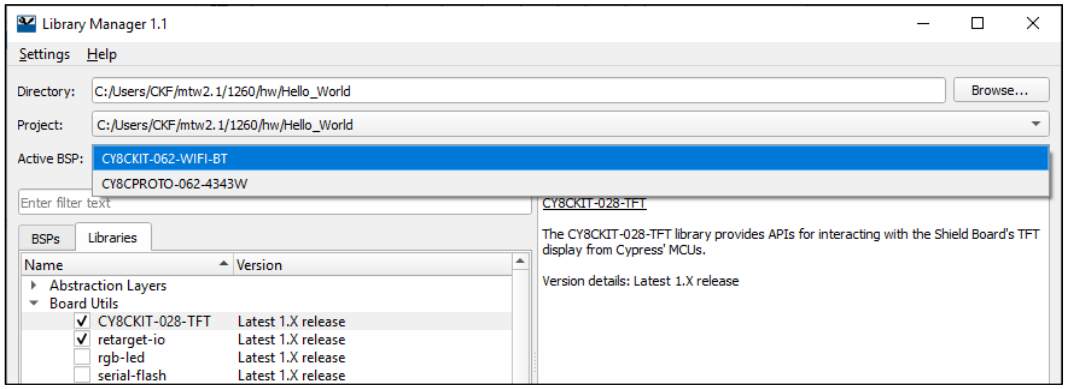
Each BSP provides HTML documentation that is specific to the selected board. It also includes an API Reference and the BSP Overview. After creating a project, there is a link to the BSP documentation in the IDE Quick Panel. As mentioned previously, this documentation is located in the BSP's "docs" directory.



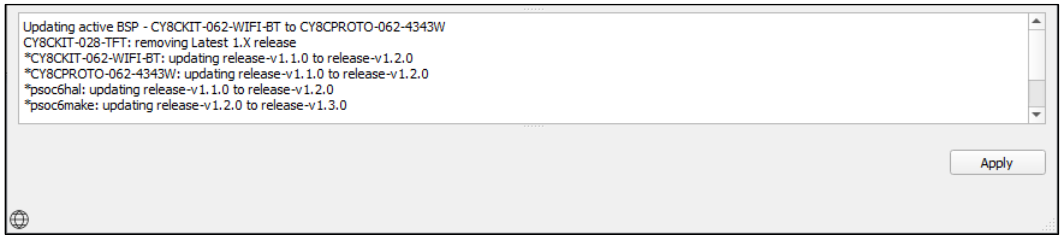


5a.8.3 Changing the BSP with the Library Manager

You can use the Library Manager to select different BSPs and different versions of BSPs to include in your application. Select the **Active BSP** to specify which one to use currently.



The console displays the changes to be made. Click **Apply** to update your application.



5a.8.4 Selecting a BSP by editing the Makefile

To specify a different BSP by editing the Makefile, update the TARGET value to the desired board.

```
27 ################################################################################
28 # Basic Configuration
29 ################################################################################
30
31 # Target board/hardware
32 TARGET=CY8CPROTO-062-4343W
```

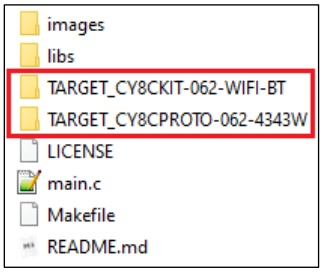
IMPORTANT: This will change the project being built by the IDE, but it will **NOT** update the Launches in the Quick Panel in the Eclipse IDE, so it does not affect which hex file is copied to the kit. Therefore, it is recommended to use the Library Manger's Active BSP selection from inside the Eclipse IDE to change the target board if you are using the IDE to program/debug. If you do edit the Makefile, there is a link in the Quick Panel that says “Generate Launches for <app-name>“ that you can use to regenerate them.

5a.8.5 Modifying the BSP Configuration (e.g. design.modus) for a Single Application

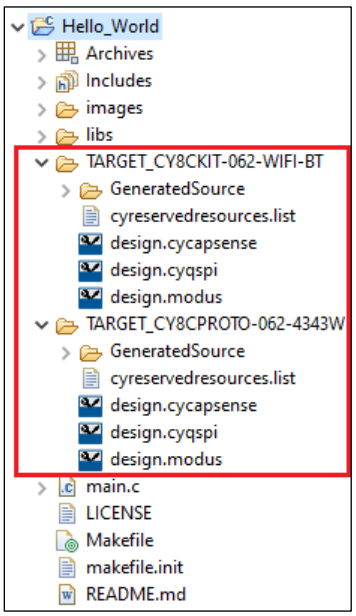
If you want to modify the BSP configuration for a single application (such as different pin or peripheral settings), it is preferable to NOT modify the BSP directly since that results in changes to the BSP library which would prevent you from updating the repository in the future. Instead, there is a mechanism to use a custom set of configuration files in an application. The steps are:

1. Create a directory for each target that you want to support at the top-level directory in your application. The directory name must be TARGET_(board_name). For example, TARGET_CY8CPROTO-062-4343W.

   Remember that the build system automatically includes all the source files inside a directory that begin with TARGET_ followed by the target name for compilation when that target is specified in the application make file. The file structure appears as follows. Here, the BSP_DESIGN_MODUS component is overridden for two targets: CY8CPROTO-062-4343W and CY8CKIT-062S2-43012.



2. Copy the design.modus file and other configuration files (i.e. everything from inside the BSP's COMPONENT_BSP_DESIGN_MODUS folder) inside this new directory and customize as required. When you save the changes in the design.modus file, the source files are generated and placed under the GeneratedSource directory. The file structure appears as follows:



3. In the makefile, add the following lines:

```
DISABLE_COMPONENTS += BSP_DESIGN_MODUS
COMPONENTS += CUSTOM_DESIGN_MODUS
```

   The first line disables the configuration files from the BSP. The second line is required so that the init_cycfg_all function is still called from the cybsp_init function. The init_cycfg_all function is used to initialize the hardware that was setup in the configuration files.
   Note that the makefile already contains blank COMPONENTS and DISABLE_COMPONENTS lines where you can add the appropriate names.

When you first create a custom configuration for an application, the Quick Panel entry to launch the Device Configurator may still open the design.modus file instead of the custom file. Therefore, it is safest to open the Device Configurator by double-clicking on the design.modus file in your CUSTOM_DESIGN_MODUS directory. It is also a good idea to verify in the banner that you have opened the correct copy. Once you switch between different applications in your workspace or quit/restart the IDE, the Quick Panel should pick up the correct file.

5a.8.6 Creating your Own BSP

If you want to change more than just the configuration from the COMPONENT_BSP_DESIGN_MODUS folder (such as for your own custom hardware or for different linker options), you can create a full BSP based on an existing one. To create your own board, do the following:

1. Locate the closest-matching BSP to your custom BSP and set that as the default TARGET for the application in the makefile.
2. Run the `make bsp` target, specifying the new board name by passing the value to the TARGET_GEN variable. Optionally, specify the new device (DEVICE_GEN) and additional devices (ADDITIONAL_DEVICES_GEN). For example:

```
make bsp TARGET_GEN=MyBSP DEVICE_GEN=CY8C624ABZI-D44
ADDITIONAL_DEVICES_GEN=CYW4343WKUBG
```

   This will create a new BSP with the provided name at the top of the application project. It will also automatically copy the relevant startup and linker scripts based on the MPN specified by DEVICE_GEN into the newly created BSP.

   - If there were any issues with the new device's configuration, open the Device Configurator to address them.
   - The BSP used as your starting point may have library references (for example, capsense.lib or udb-sdio-whd.lib) that are not needed by your custom BSP. These can be deleted from the BSP.
3. Define or update the alias for pins in the cybsp_types.h file.
4. Customize the design.modus file and other configuration files with new settings for clocks, power supplies, and peripherals as required.
5. Update the make TARGET variable to point to your new BSP. If you're starting from a Cypress provided example project, you will find this in the makefile file in the root of your project.

6. If using an IDE, regenerate the configuration settings to reflect the new BSP. Pick the appropriate command(s) for the IDE(s) that are being used. For example: `make vscode`

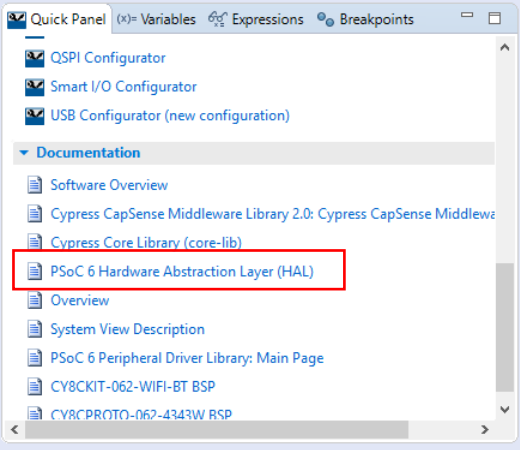
Note: The full list of IDEs is dependent on the make build system being used. Use make help to see all supported IDE make targets.

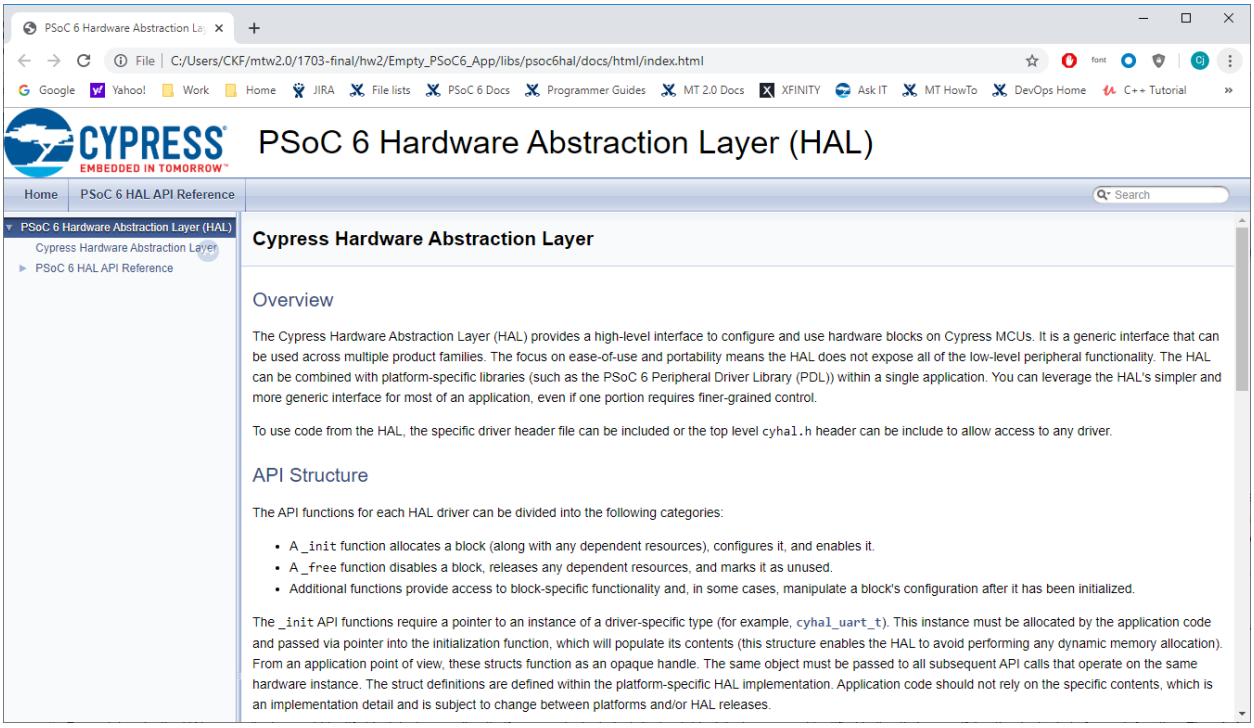
If you want to re-use a custom BSP on multiple applications, you can copy it into each application, or you can put it into a git repo so that you can use it just like any other BSP during application creation.

5a.9 HAL

The Cypress Hardware Abstraction Layer (HAL) provides a high-level interface to configure and use hardware blocks on Cypress MCUs. It is a generic interface that can be used across multiple product families. The focus on ease-of-use and portability means the HAL does not expose all of the low-level peripheral functionality. The HAL can be combined with platform-specific libraries (such as the PSoC 6 Peripheral Driver Library (PDL)) within a single application. You can leverage the HAL's simpler and more generic interface for most of an application, even if one portion requires finer-grained control.

After creating a PSoC 6 project, there is a link to the HAL documentation in the Quick Panel under "Documentation."

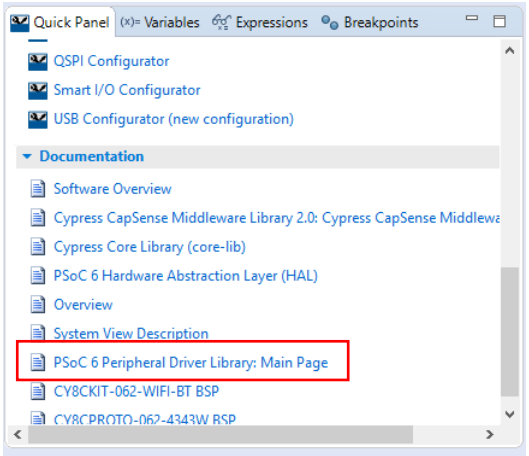


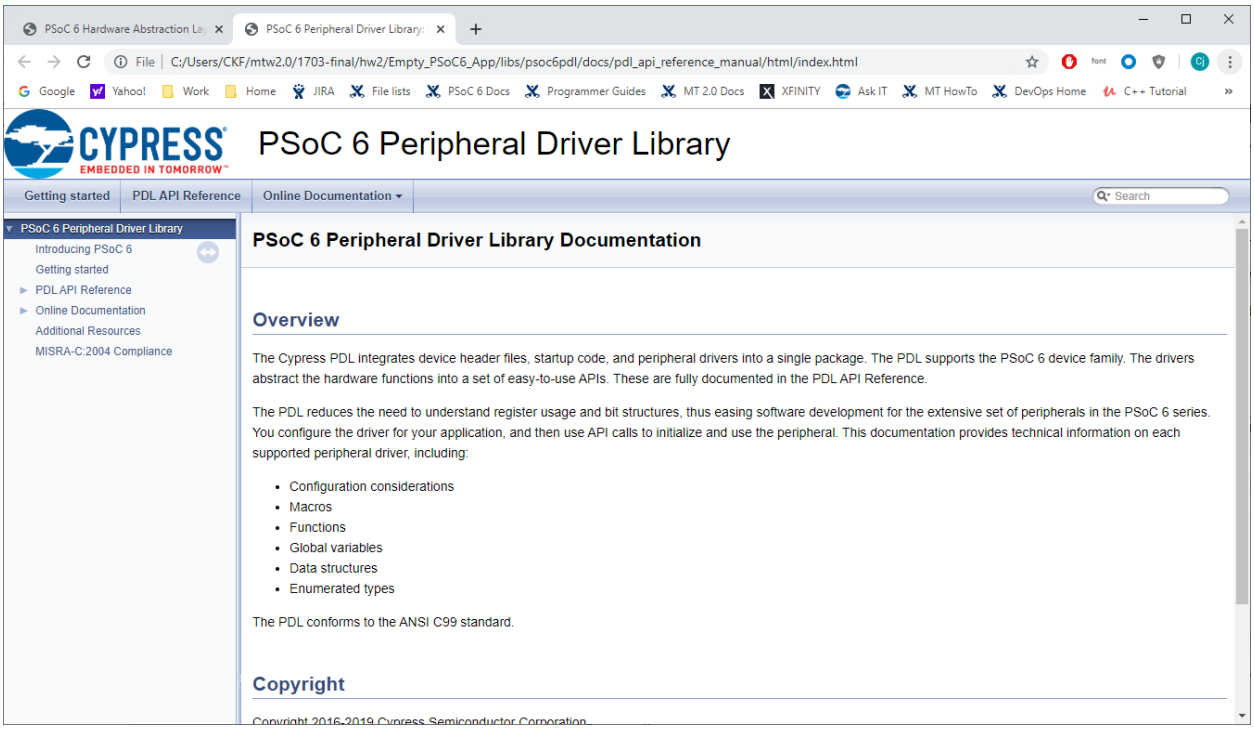


5a.10 PDL

The Peripheral Driver Library (PDL) integrates device header files, startup code, and peripheral drivers into a single package. The PDL reduces the need to understand register usage and bit structures, thus easing software development for the extensive set of peripherals in the PSoC 6 series.

After creating a PSoC 6 project, there is a link to the PDL documentation in the Quick Panel under "Documentation."





5a.11 Manifests

Manifests are XML files that tell the Project Creator and Library Manager how to discover the list of available boards, example projects, and libraries. There are several manifest files.

- The “super manifest” file contains a list of URLs that software uses to find the board, code example, and middleware manifest files.
- The “app-manifest” file contains a list of all code examples that should be made available to the user.
- The “board-manifest” file contains a list of the boards that should be presented to the user in the new project creation tool as well as the list of BSP packages that are presented in the Library Manager tool.
- The “middleware-manifest” file contains a list of the available middleware (libraries). Each middleware item can have one or more versions of that middleware available.

To use your own examples, BSPs, and middleware, you need to create manifest files for your content and a super-manifest that points to your manifest files.

Once you have the files created in a Git repo, place a manifest.loc file in your <user_home>/.modustoolbox directory that specifies the location of your custom super-manifest file (which in turn points to your custom manifest files). For example, a manifest.loc file may have:

```
# This points to the IOT Expert template set
https://github.com/iotexpert/mtb2-iotexpert-manifests/raw/master/iotexpert-super-manifest.xml
```

Note that you can point to local super-manifest and manifest files by using file:/// with the path instead of https://. For example:

```
file:///C:/MyManfests/my-super-manifest.xml
```

To see examples of the syntax of super-manifest and manifest files, you can look at the Cypress provided files on GitHub:

- Super Manifest: https://github.com/cypresssemiconductorco/mtb-super-manifest
- App Manifest: https://github.com/cypresssemiconductorco/mtb-ce-manifest
- Board Manifest: https://github.com/cypresssemiconductorco/mtb-bsp-manifest
- Middleware Manifest: https://github.com/cypresssemiconductorco/mtb-mw-manifest

5a.12 Network Proxy Settings

Depending on the location of this class you may need to set up your network proxy settings for ModusToolbox - specifically the Project Creator and Library Manager since they both require internet access. Proxy settings will vary from site to site.

A list of Cypress proxies can be found at: https://itsm.cypress.com/solutions/726105.portal. If you are at a non-Cypress site, contact that site's IT department for the proxy name and port.

If you have environment variables for http_proxy and https_proxy (for example, to get Mbed to work), those same variables will work for ModusToolbox. However, there is a simpler way to enable/disable the proxy settings in the Project Creator and Library Manager tools that doesn't require you to create/delete the environment variables whenever you want to switch between a network that requires a proxy and a second network that does not require a proxy.

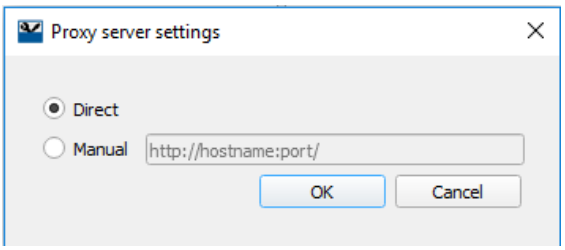
When you run the Project Creator or Library Manager, the first thing it does is look for a remote manifest file. If that file isn't found, you will not be able to go forward. In some cases, it may find the manifest but then fail during the project creation step (during git clone). If either of those errors occur, it is likely due to one of these reasons:

- You are not connected to the internet. In this case, you can choose to use offline content if you have previously downloaded it. Offline content is discussed in the next section.
- You may have incorrect proxy settings (or no proxy settings).

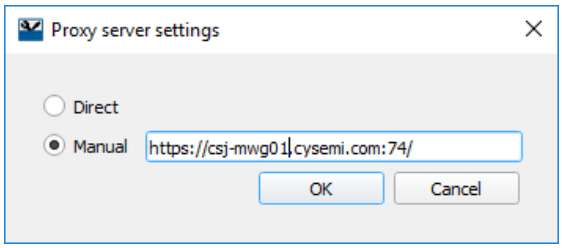
To view/set proxy settings in the Project Creator or Library Manager, use the menu option **Settings > Proxy Settings…**

If your network doesn't require a proxy, choose “Direct”. If your network requires a proxy choose “Manual”. Enter the server name and port in the format http:/hostname:port/.

No Proxy



Cypress San Jose Proxy



Once you set a proxy server, you can enable/disable it just by selecting Manual or Direct. There is no need to re-enter the proxy server every time you connect to a network requiring that proxy server. The tool will remember your last server name.

Note that "Direct" will also work if you have the http_proxy and https_proxy environment variables set so if you prefer to use the environment variables, just use "Direct" and create/delete the variables depending on whether your site needs the proxy or not.

The settings made in either the Project Creator or Library Manager apply to the other.

5a.13 Offline Content

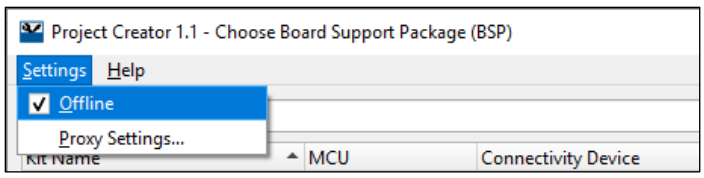
Beginning with ModusToolbox version 2.1, Cypress provides a zipped-up bundle of the GitHub repos to allow you to work offline, such as on an airplane or if for some reason you don't have access to GitHub. To set this up, you must have access to cypress.com in order to download the zip file. After that, you can work offline.

Go to https://www.cypress.com/modustoolbox-offline-content and download the modusttoolbox-offline-content-x.x.x.<build>.zip file.

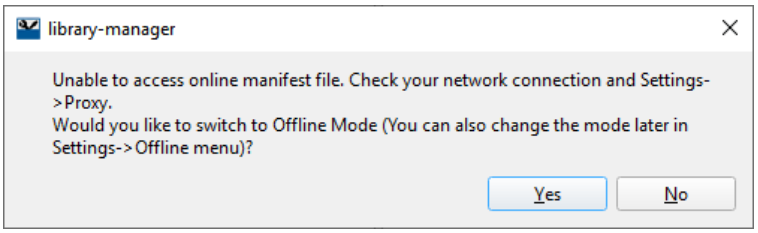
Extract the "offline" directory into the root location where you have ModusToolbox installed into a hidden directory named ".modustoolbox". In most cases, this is in your User Home directory. After extracting, the path should be like this:

C:\Users\XYZ\.modustoolbox\offline

To use the offline content, toggle the setting in the Project Creator and Library Manager tools, as follows:



Also, if you try to open these tools without a connection, a message will ask you to switch to Offline Mode.



5a.14 Cypress Configurators

ModusToolbox software provides graphical applications called configurators that make it easier to configure a hardware block. For example, instead of having to search through all the documentation to configure a serial communication block as a UART with a desired configuration, open the appropriate configurator to set the baud rate, parity, stop bits, etc. Upon saving the hardware configuration, the tool generates the C code to initialize the hardware with the desired configuration.

Many configurators do not apply to all types of projects. So, the available configurators depend on the project/application you have selected in the Project Explorer. Configurators are independent of each other, but they can be used together to provide flexible configuration options. They can be used stand alone, in conjunction with other configurators, or within a complete IDE. Everything is bundled together as part of the installation. Each configurator provides a separate guide, available from the configurator's Help menu. Configurators perform tasks such as:

- Displaying a user interface for editing parameters
- Setting up connections such as pins and clocks for a peripheral
- Generating code to configure middleware

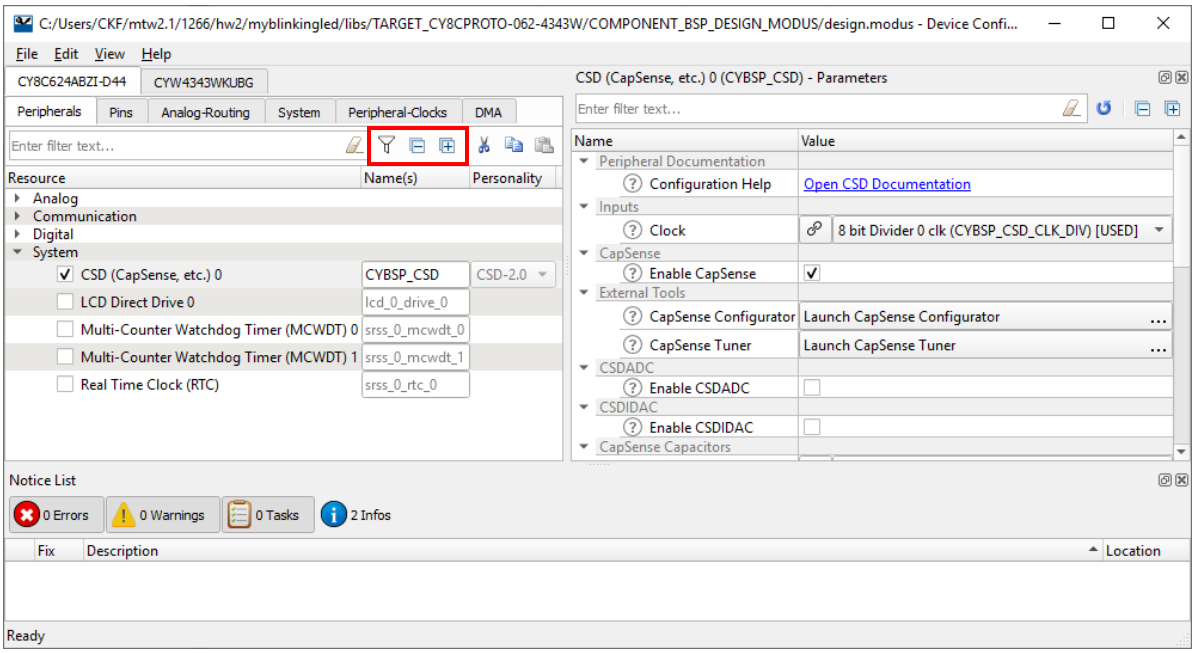
Configurators are divided into two types: Board Support Package (BSP) Configurators and Library Configurators.

5a.14.1 BSP Configurators

BSP Configurators are closely related to the hardware resources on the board such as CapSense sensors, external Flash memory, etc. These are typically provided inside the BSP hierarchy, although they can be customized for a given application as was described earlier.

Device Configurator

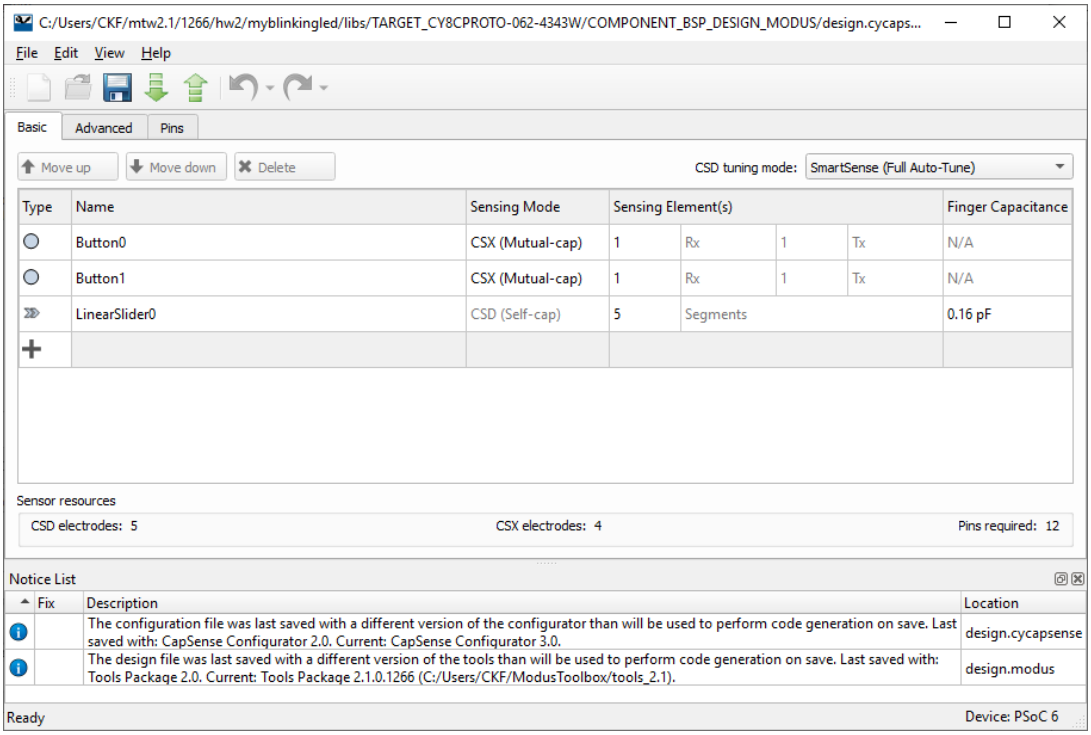
Set up the system (platform) functions such as pins, interrupts, clocks, and DMA, as well as the basic peripherals, including UART, Timer, etc.

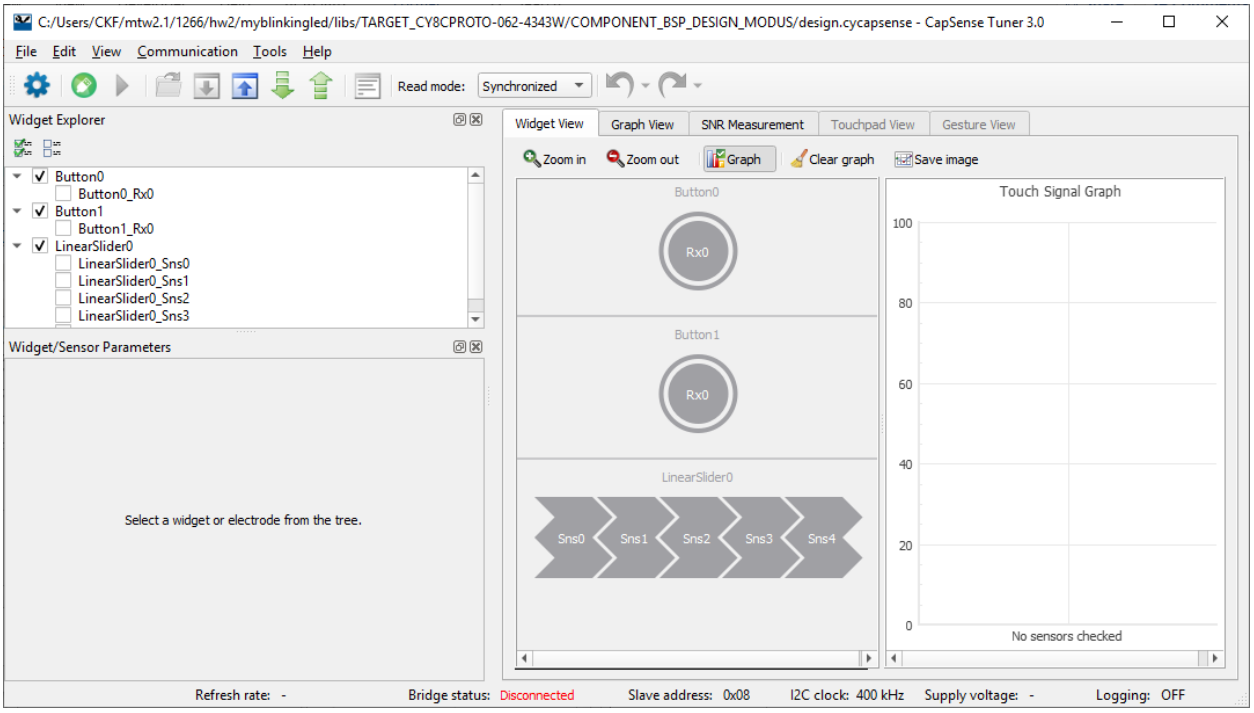


Hint: The + and - buttons can be used to expand/contract all categories and the filter button can be used to show only items that are selected. This is particularly useful on the Pins tab since the list of pins is sometimes quite long.

CapSense Configurator/Tuner

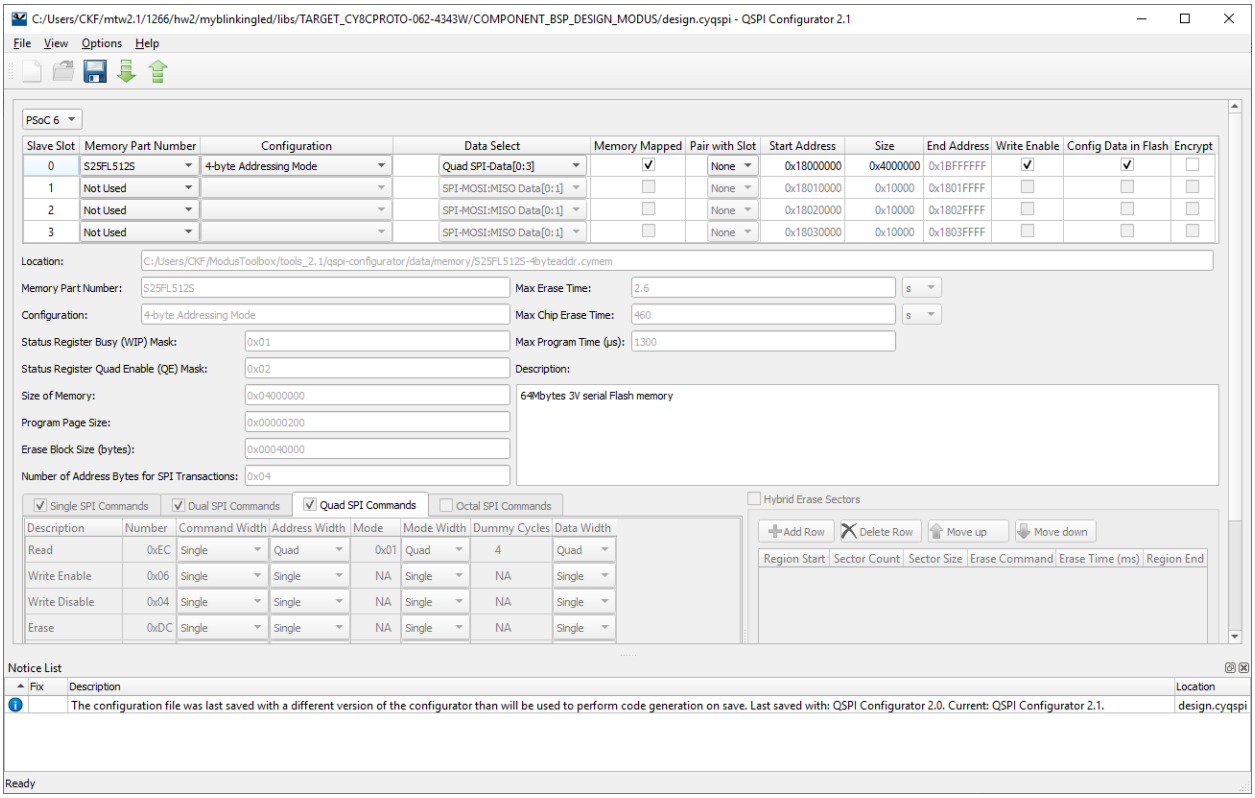
Configure CapSense hardware and generate the required firmware. This includes tasks such as mapping pins to sensors and how the sensors are scanned.





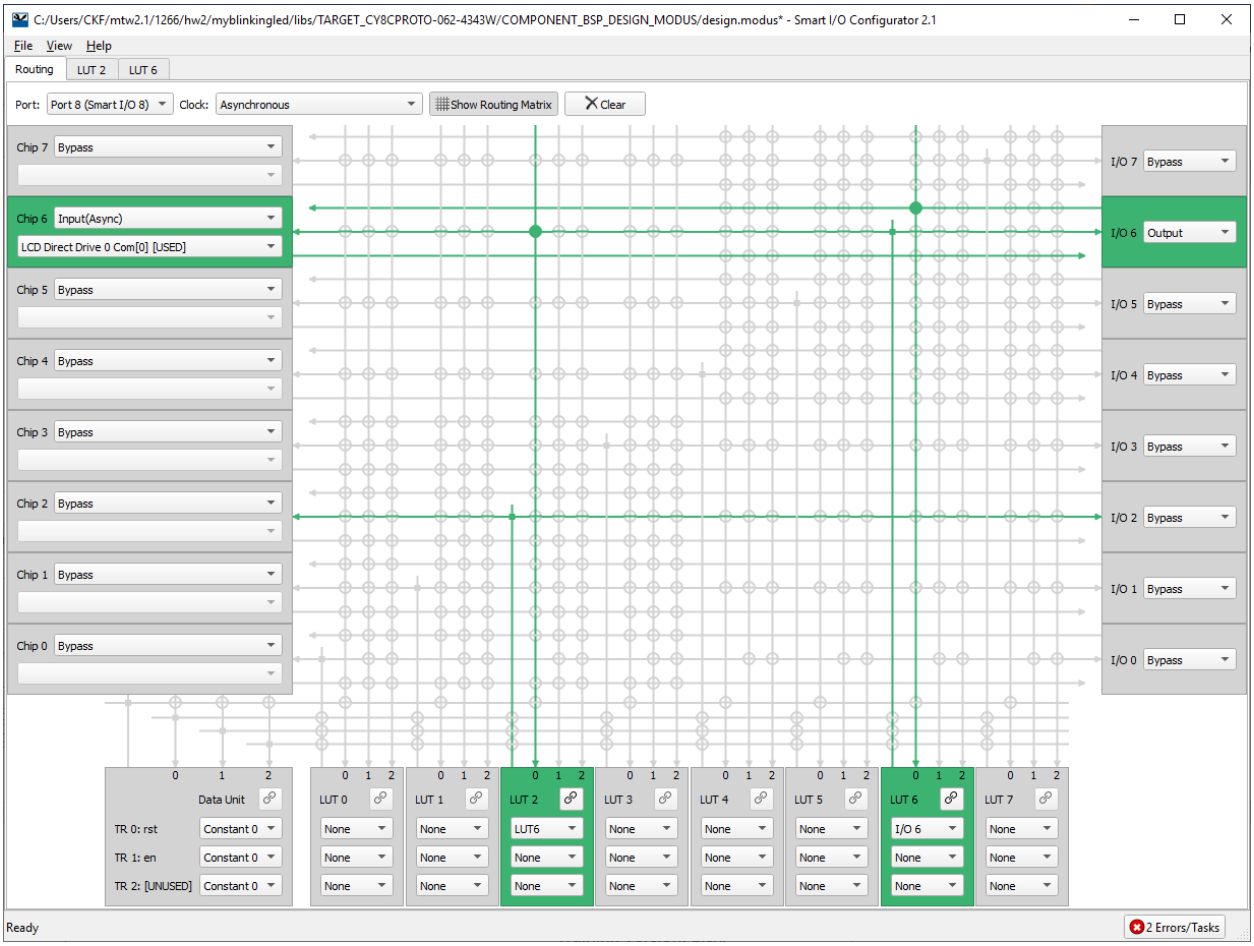
QSPI Configurator

Configure external memory and generate the required firmware. This includes defining and configuring what external memories are being communicated with.



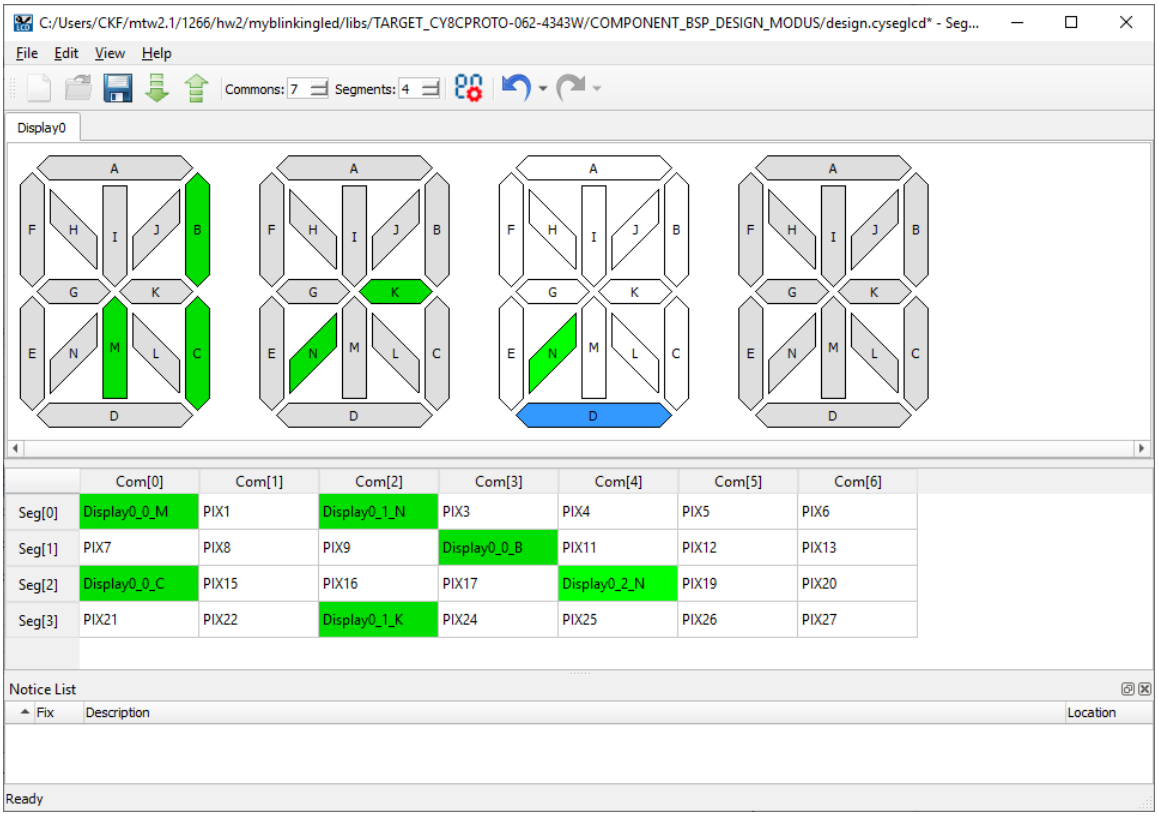
Smart I/O™ Configurator

Configure the Smart I/O block, which adds programmable logic to an I/O port.



SegLCD Configurator

Configure LCD displays. This configuration defines a matrix Seg LCD connection and allows you to setup the connections and easily write to the display.

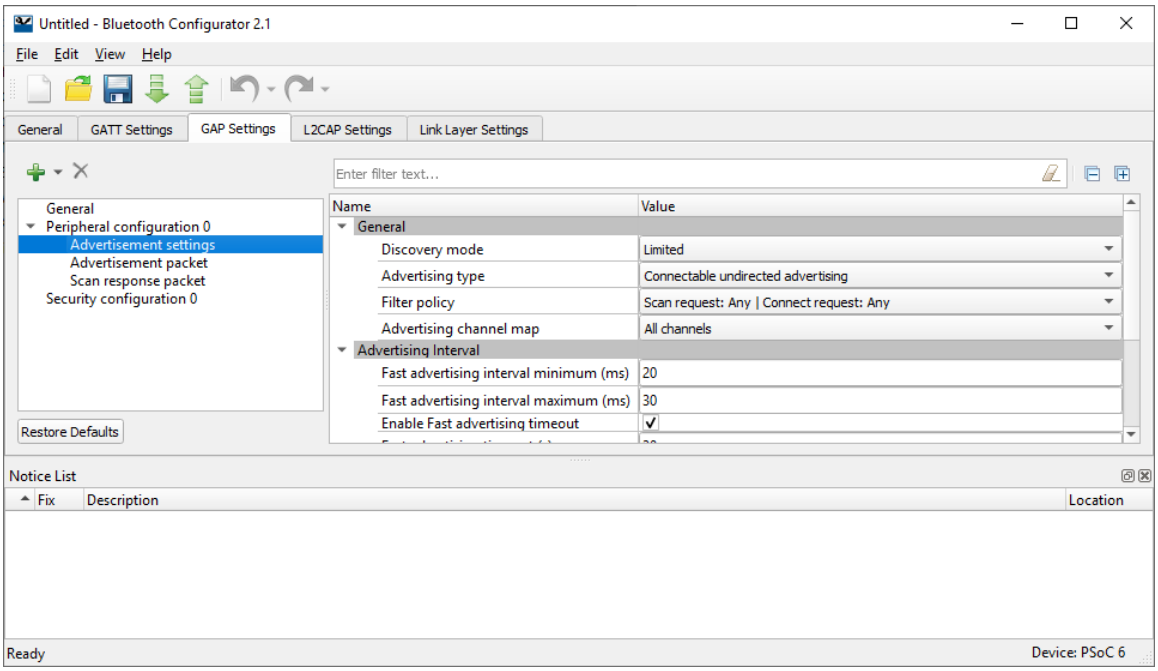


5a.14.2 Library configurators

Library Configurators are used to setup and configure middleware libraries. The files for these are contained in the project hierarchy rather than inside the BSP.

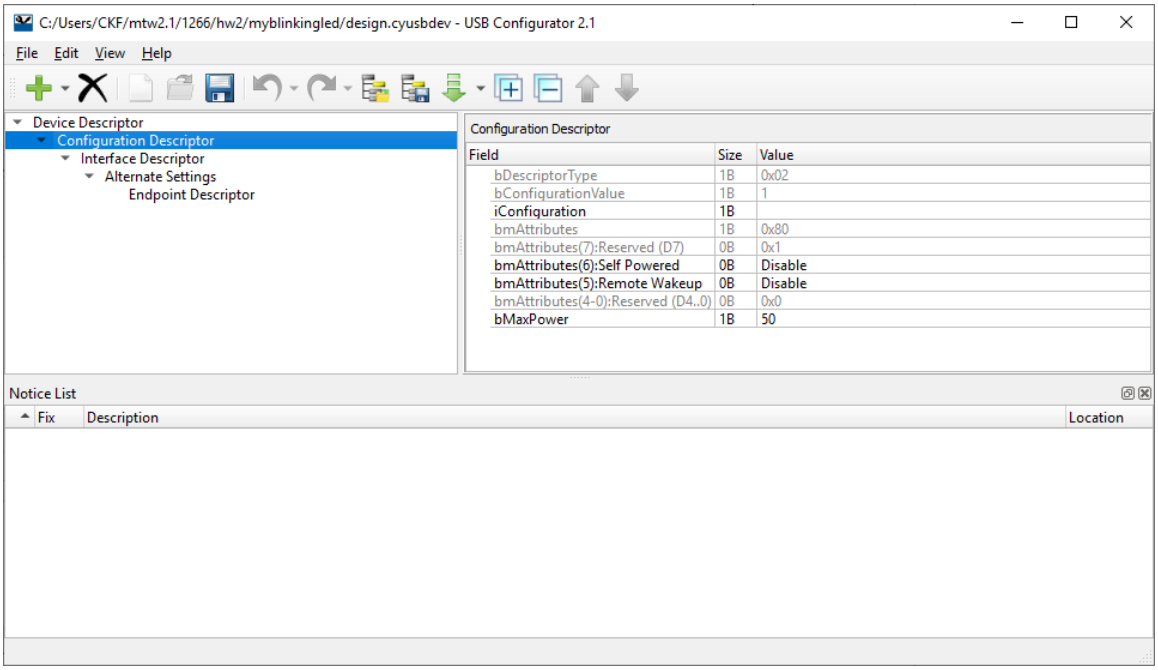
Bluetooth Configurator

Configure Bluetooth settings. This includes options for specifying what services and profiles to use and what features to offer by creating SDP and/or GATT databases in generated code. This configurator supports both PSoC MCU and WICED Bluetooth applications.



USB Configurator

Configure USB settings and generate the required firmware. This includes options for defining the 'Device' Descriptor and Settings.



5a.15 FreeRTOS

Information on how to include FreeRTOS in an application can be found in the RTOS chapter. The short summary is:

1. Add the freertos library.
2. Copy the file FreeRTOSConfig.h from the directory libs/freertos/Source/portable to your application's root directory.
3. Edit the settings in FreeRTOSConfig.h to suit your application's needs.

Information on using low power in FreeRTOS designs can be found in the PSoC 6 Low Power chapter.

5a.16 Dual Core Designs

TBD

5a.17 Exercises (Part 2)

The following exercises can be completed from the command line or within the IDE. Feel free to use the method that feels most natural, interesting, educational, challenging (or easy) – it's your choice! You can also change your mind half-way through if you wish to practice using both approaches.

5a.17.1 Exercise 3: Modify the blinking LED to use the Green LED

1. Open the main.c file from your previous project in an editor.
2. Locate this line of code:

```
cyhal_gpio_init(CYBSP_USER_LED1,CYHAL_GPIO_DIR_OUTPUT,CYHAL_GPIO_DRIVE_STRON
G, CYBSP_LED_STATE_OFF);
```

3. Change the LED that you want to blink from `CYBSP_USER_LED1` to `CYBSP_LED_RGB_GREEN`, which is the green LED in the RGB LED (this is defined in the BSP file cybsp_types.h).
4. Locate this line of code and make the same change so that the LED will blink:

```
cyhal_gpio_toggle(CYBSP_LED_RGB_GREEN);
```

5. Build the project and program the board.

5a.17.2 Exercise 4: Modify the project to Blink Green/Red using the HAL

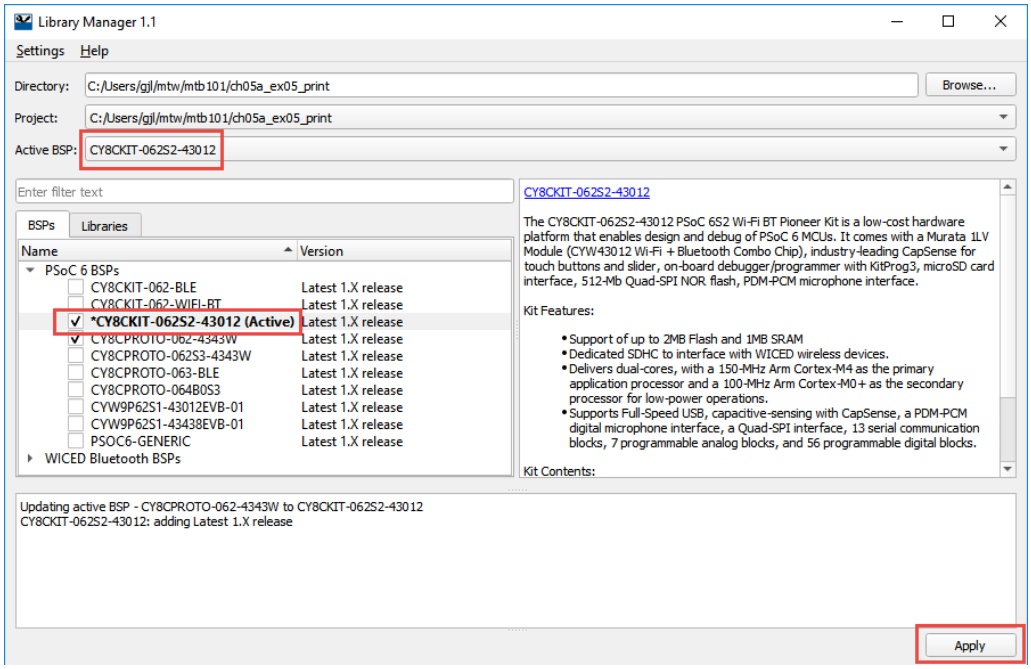
1. Make a copy of the line of code that initializes the `CYBSP_LED_RGB_GREEN`.
2. Change the LED to `CYBSP_LED_RGB_RED` so you are initializing the red LED from the RGB LED as well as the green LED.
3. Change the final argument to cyhal_gpio_init() so that `CYBSP_LED_RGB_RED` is initialized in the ON state while `CYBSP_LED_RGB_GREEN` is OFF.
4. Make a copy of the line of code that toggles the green LED and edit it to toggle the red LED.
5. Build the project program the board.

5a.17.3 Exercise 5: Print a message using the retarget-io library

1. Create a new project for the CY8CKIT-062S2-43012 kit with the "Empty PSoC6 App' template. Name the project **ch05a_ex05_print**.

   Note: If you are working from the command line, you will need to add the BSP for your kit. You could just do a manual `git clone` of the repo and then run `make getlibs`, but it is much easier to use `make modlibs` to run the Library Manager, so I'll do that instead. I can set the active BSP from the Library Manager at the same time, so I don't have to specify the target on the command line when building/programming or editing the target in the makefile. Here are the steps:

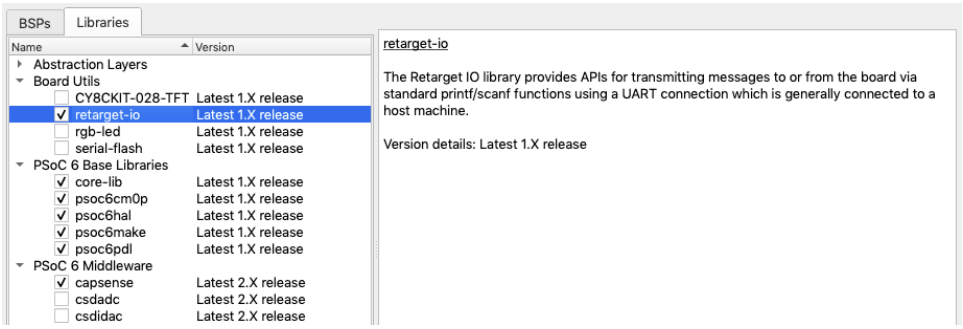
   ```
   git clone https://github.com/cypresssemiconductorco/mtb-example-psoc6-empty-app
   mv mtb-example-psoc6-empty-app ch05a_ex05_print
   cd ch05a_ex05_print
   make modlibs
   ```



2. Launch the Library Manager to add a library to our application.
   - In the IDE you can do this from the Quick Panel (Tools) or use the right-mouse menu on the project root node and go to **ModusToolbox > Library Manager**.
   - From the command line just use `make modlibs` again.

3. Observe the available BSPs, then switch to the **Libraries** tab.

4. Find the retarget-io library, check the box, and click **Apply** to add the library to your application.



5. Back in the IDE (or using your favorite editor from the command line), open main.c and add the following include file to tell the compiler you wish to use the retarget-io functions:

```
#include "cy_retarget_io.h"
```

6. Add the following include file to tell the compiler you wish to use the printf() function.

```
#include <stdio.h>
```

7. In the main() function, add the following code to initialize the UART and associate with the retarget-io library.

```
result = cy_retarget_io_init(CYBSP_DEBUG_UART_TX, CYBSP_DEBUG_UART_RX, \
CY_RETARGET_IO_BAUDRATE);
CY_ASSERT(result == CY_RSLT_SUCCESS);
```

Make sure you put this AFTER the cybsp_init function has been called. The BSP init should just about always be the first thing you do.

8. Before the forever loop, use printf() to write “PSoC Rocks!\r\n” to the UART.

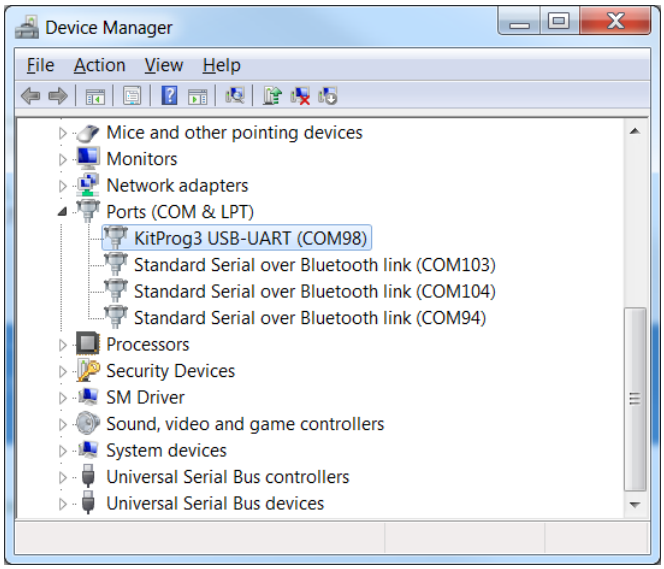
If you don't want to include \r\n every time, you can just use \n. In that case, you can either setup your terminal window to automatically add a \r for every \n or you can use a handy feature in the retarget-io library to do add it for you. Look at the retarget-io library API documentation to learn how to do that.

If you forget to put \n, the message won't be printed until the buffer fills up. In this example that will never happen, so don't forget to include \n.

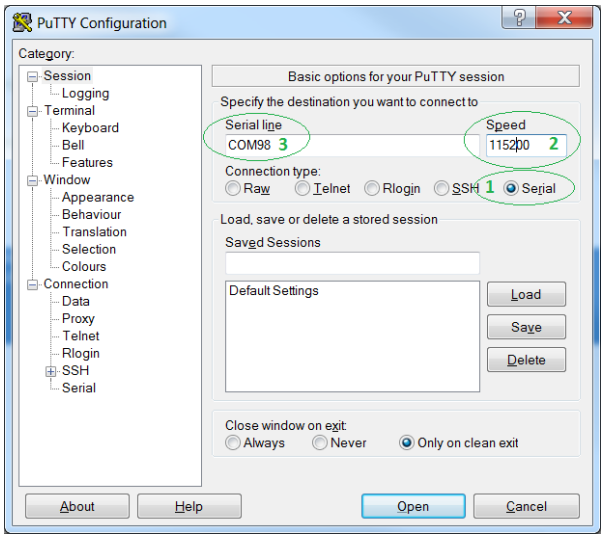
9. Build the project and program the board.

10. From the Windows Search, type “device manager” and launch that utility.

11. Look in the “Ports (COM&LPT)” directory for your board’s KitProg3. Note the COM port number.



12. From Windows, launch the PuTTY application (or your favorite terminal emulator).
13. Specify a Serial session, at a Speed of 115200 (baud rate), using the Serial Line from the device Manager (COM PORT).



14. Observe the output from your kit when you reset it.

5a.17.4 Exercise 6: Control the RGB LED with a Library

1. Return to the Library Manager and follow the same process to add the rgb-led library to your project.
2. In the IDE, at the top of main.c, and add the following include file to tell the compiler you wish to use the rgb_led functions:

```
#include "cy_rgb_led.h"
```

3. Add this macro – it translates the name of a #define into a string:

```
#define DEFINE_TO_STRING(macro) (#macro)
```

4. In the main function, add this code to initialize the RGB LED. It defines the LED pins (defined for you in the BSP) and sets the polarity:

```
result = cy_rgb_led_init(CYBSP_LED_RGB_RED, CYBSP_LED_RGB_GREEN,
CYBSP_LED_RGB_BLUE, CY_RGB_LED_ACTIVE_LOW);

if (result != CY_RSLT_SUCCESS)
{
CY_ASSERT(0);
}
```

5. In the forever loop add this code to make the LED yellow for a second.

```
printf( "Color is %s\r\n", DEFINE_TO_STRING(CY_RGB_LED_COLOR_YELLOW));
cy_rgb_led_set_color(CY_RGB_LED_COLOR_YELLOW, CY_RGB_LED_MAX_BRIGHTNESS);
Cy_SysLib_Delay( 1000 );
```

6. Make copies of those three lines and turn the LED PURPLE and CYAN.
7. Build the project program the board. Observe the printed output in the terminal and the LED behavior.

5a.17.5 Exercise 7: CapSense Buttons and Slider

Build the example code

1. Create the CapSense Buttons and Slider starter application using the IDE or command line. Name the application **ch05a_ex07_capsense**.

   Note: If you are working from the command line you will need to add the BSP for your kit.
2. Build the project and program the board.

Run the CapSense Tuner

1. Launch the CapSense Tuner through the IDE or in stand-alone mode. Note that, in the latter case, you will need to manually open the CapSense configuration file:

   libs/TARGET_ CY8CKIT-062S2-43012/COMPONENT_BSP_DESIGN_MODUS/design.cycapsense

   If you are using the command line, you can start the CapSense Tuner with this command:

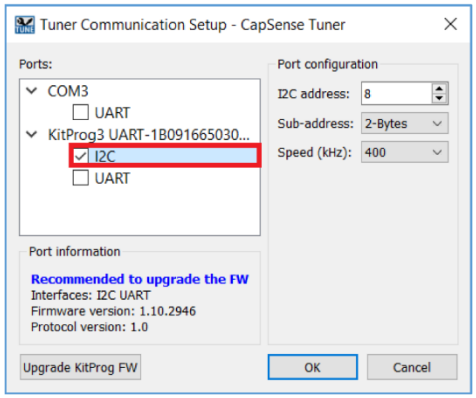
   ```
   make open CY_OPEN_TYPE=capsense-tuner
   ```

2. In the Tuner, click **Tools > Tuner Communication Setup** or press [F10].

   Note: If you have a terminal emulator connected to the serial port this will interfere with the tuner connection so close the terminal window if you have connection difficulties.

3. In the dialog, select I2C under KitProg and configure as follows:

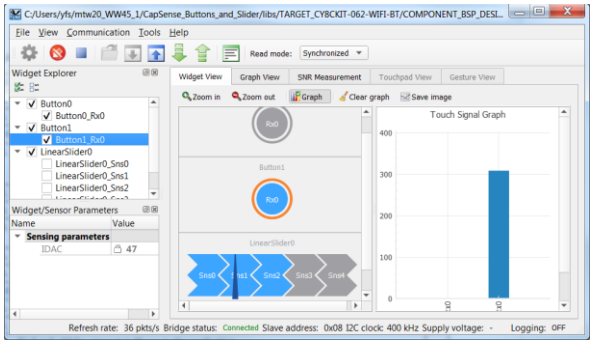
4. Check the parameters and close the dialog.
   - I2C Address: 8
   - Sub-address: 2-Bytes
   - Speed (kHz): 400

5. In the Tuner, click the **Connect** button to establish communication with the target (F4).

6. Click the **Start** button to begin collecting tuning data (F5).

7. Verify that tuning is operational by touching widgets and observing the data in the Widget and Graph view.

   Note: You must enable each sensor in the Widget Explorer window to use the graph view.



8. Stop collecting data (Shift+F5).
9. Close the connection (Shift+F4).

5a.17.6 Exercise 8: Write a message to the TFT shield

This project displays a message on the CY8CKIT-028-TFT shield, which is attached to the CY8CKIT-062S2-43012 board.

1. Create an Empty_PSoC6_App project for the CY8CKIT-062S2-43012 kit. Name the application **ch05a_ex08_tft**.

   Note: If you are working from the command line, you will need to add the BSP for your kit.

2. Using the Library Manager, add the CY8CKIT-028-TFT library from **Board Utils** and the emwin library from **PSoC 6 Middleware**.
3. Click **Apply** and close the Library Manager.
4. Open the project Makefile to add emwin configuration for bare metal without touchscreen:

   ```
   COMPONENTS+=EMWIN_NOSNTS
   ```

5. In the main.c file, change the code to match the following:

```
#include "cy_pdl.h"
#include "cyhal.h"
#include "cybsp.h"
#include "GUI.h"

int main(void)
{
    cy_rslt_t result;

    /* Initialize the device and board peripherals */
    result = cybsp_init() ;
    if (result != CY_RSLT_SUCCESS)
    {
        CY_ASSERT(0);
    }

    __enable_irq();

    GUI_Init();
       GUI_SetColor(GUI_WHITE);
       GUI_SetBkColor(GUI_BLACK);
       GUI_SetFont(GUI_FONT_32B_1);
       GUI_SetTextAlign(GUI_TA_CENTER);
       /* Change this text as appropriate */
       GUI_DispStringAt("I feel good!",
GUI_GetScreenSizeX()/2,GUI_GetScreenSizeY()/2 - GUI_GetFontSizeY()/2);

    for(;;)
    {
    }
}
```

6. Build and program your project.

   Observe the message displayed on TFT.

5a.17.7 Exercise 9: Use an RTOS to Blink LEDs

1. Create another new project for the CY8CKIT-062S2-43012 kit with the "Empty PSoC6 App' template and call it **ch05a_ex09_blink_rtos**.

   Note: If you are working from the command line you will need to add the BSP for your kit.

2. Use the Library Manager to add the freertos library.

3. Close the Library Manager.

4. Open main.c and add the FreeRTOS headers.

   ```
   #include "FreeRTOS.h"
   #include "task.h"
   ```

   Note: FreeRTOS.h is needed in all files that make RTOS calls and tasks.h, mutex.h, semphr.h, and so on are used when code accesses resources of that type.

5. Copy the file file **FreeRTOSConfig.h** from libs/freertos/Source/portable to the top directory of your application.

   This file contains application specific FreeRTOS configuration information. The build system will automatically include the file that is found at the top of directory hierarchy.

   Note: If you want to put the file in a different location, you must specify the path to the directory relative to the application's top-level directory in the INCLUDES variable in the Makefile.

6. Open the FreeRTOSConfig.h file from the top application directory and remove the line that says:

   ```
   #warning This is a template. Copy this file to your project and remove this
   line. Refer to FreeRTOS README.md for usage details.
   ```

7. Write a function to blink an LED that will be called by the RTOS when the task starts.

   Note: do not use Cy_SysLib_Delay() in a task because it will not do what you want - use vTaskDelay() instead.

   ```
   void LED1_Task(void *arg)
   {
       cyhal_gpio_init(CYBSP_USER_LED1, CYHAL_GPIO_DIR_OUTPUT,
                       CYHAL_GPIO_DRIVE_STRONG, CYBSP_LED_STATE_OFF);
       for(;;)
       {
          cyhal_gpio_toggle(CYBSP_USER_LED1);
          vTaskDelay(200);
       }
   }
   ```

8. In main() create the task and start the task scheduler.

   ```
   xTaskCreate(LED1_Task, "LED1", configMINIMAL_STACK_SIZE, 0 /* args */, 0 /*
   priority */, NULL /* handle */);
   vTaskStartScheduler();
   ```

   Note: You can remove the `for(;;)` loop since the call to `vTaskStartScheduler` will never return.

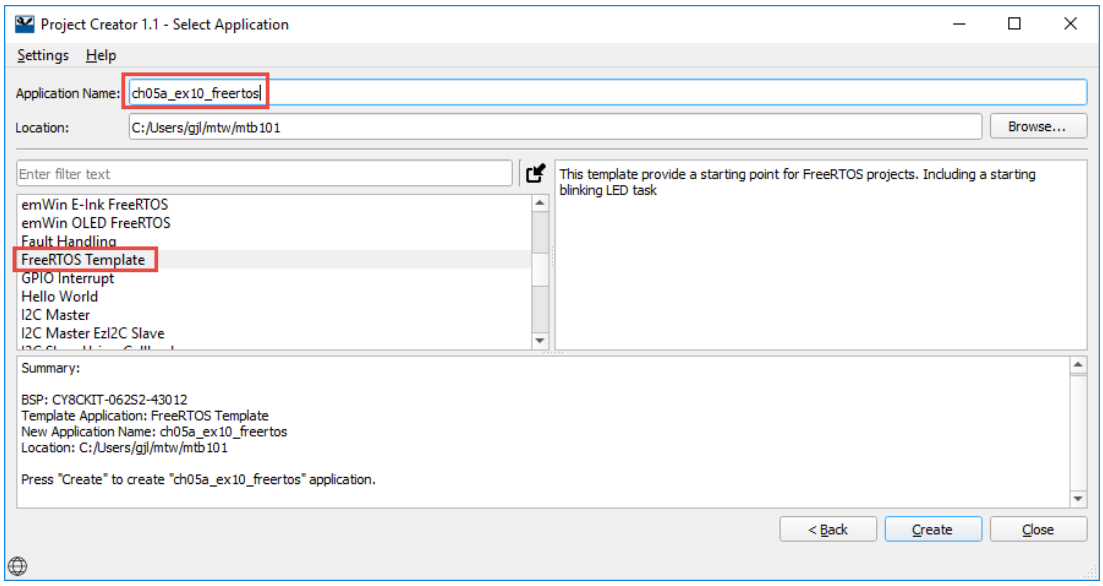
9. Build the project and program the kit. Observe the LED blinking.

10. Make a copy of the LED1_Task for LED2 and change the delay value so the LEDs blink asynchronously.
11. In main() create this task with the same priority and stack size.
12. Build the program and program the kit. Observe the LEDs blinking at different rates.

5a.17.8 Exercise 10: Use the IoT Expert FreeRTOS Template

There is a set of manifest files located at: https://github.com/iotexpert/mtb2-iotexpert-manifests. Use these manifest files to create the IoT Expert FreeRTOS Template project.

1. Copy just the manifest.loc file to your home directory in ~/.modustoolbox.
2. Create a new project using the "FreeRTOS Template" that gets picked up by the Project Creator from the iotexpert manifests. Name the project **ch05a_ex10_freertos**.



3. Build and Program.
4. What does the project do?

5. How many .lib files are in the project? What are they?

6. Make the LED blink faster.

7. Add the ntshell library.

   It is in the iotexpert GitHub site. It will now show up in the Library Manger under the category iotexpert because of the manifest files that we added.

8. Follow the instructions in the libs/middleware-ntshell/README.md file to create the ntshell task.

   Look in the section entitled “How to add to the PSoC6 SDK”.

   Note: There is a template directory in the library with default usrcmd.* files that you should copy to your project's root directory if you didn't already do that.

9. Build and Program.

10. Open a terminal emulator to test the shell. Type `help` to see a list of available commands. These are defined in the file usrcmd.c in the cmdlist array.

11. Add a new command to usrcmd.c to change the LED blink rate.

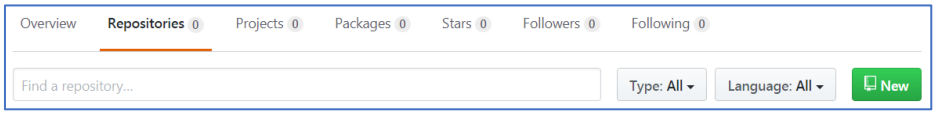
5a.17.9 Exercise 11: Create and Use a Library

For this project, create a library and add a .lib that uses the HAL to instantiate the PWM and change the brightness and color of an RGB LED.

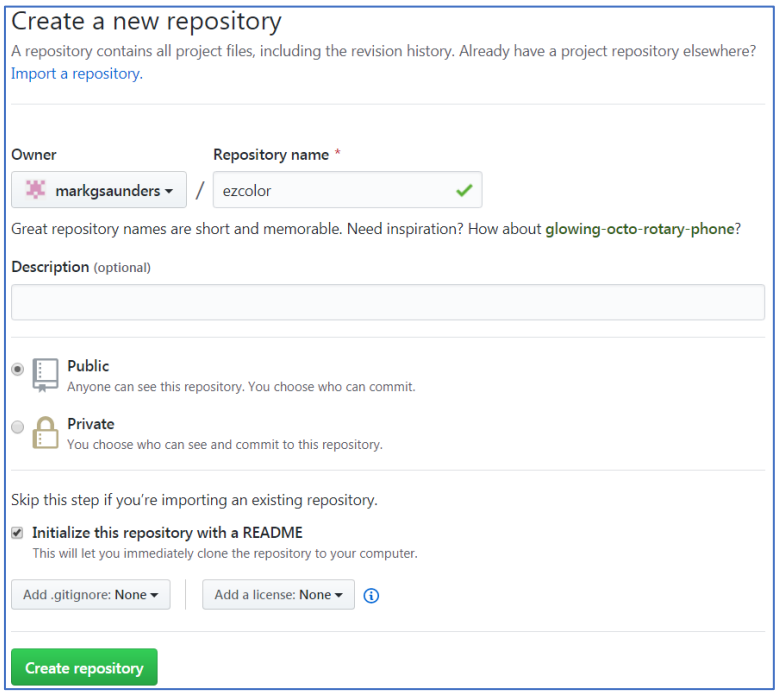
1. In a browser, go to https://www.github.com/

2. Follow the instructions to create an account or login if you already have one.

3. Click on the Repositories tab and click the green New button to create a repo.

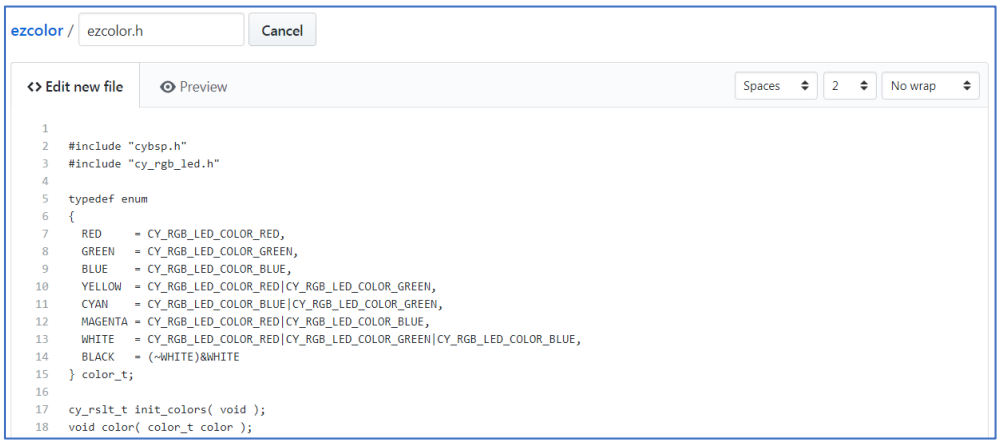


4. Give your repo a name and description - I called mine “ezcolor”. Make it public and let it create a README.md file. Press the Create Repository button.



5. Press the Create new file button
6. Name it “ezcolor.h”
7. Add these lines into the editor.

```
#include "cybsp.h"
#include "cy_rgb_led.h"
typedef enum
{
  RED     = CY_RGB_LED_COLOR_RED,
  GREEN   = CY_RGB_LED_COLOR_GREEN,
  BLUE    = CY_RGB_LED_COLOR_BLUE,
  YELLOW  = CY_RGB_LED_COLOR_RED|CY_RGB_LED_COLOR_GREEN,
  CYAN    = CY_RGB_LED_COLOR_BLUE|CY_RGB_LED_COLOR_GREEN,
  MAGENTA = CY_RGB_LED_COLOR_RED|CY_RGB_LED_COLOR_BLUE,
  WHITE   =
CY_RGB_LED_COLOR_RED|CY_RGB_LED_COLOR_GREEN|CY_RGB_LED_COLOR_BLUE,
  BLACK   = (~WHITE)&WHITE
} color_t;
cy_rslt_t init_colors( void );
void color( color_t color );
```



8. Press the **Commit new file** button.

9. Repeat the process to create “ezcolor.c”. Paste this code into the editor.

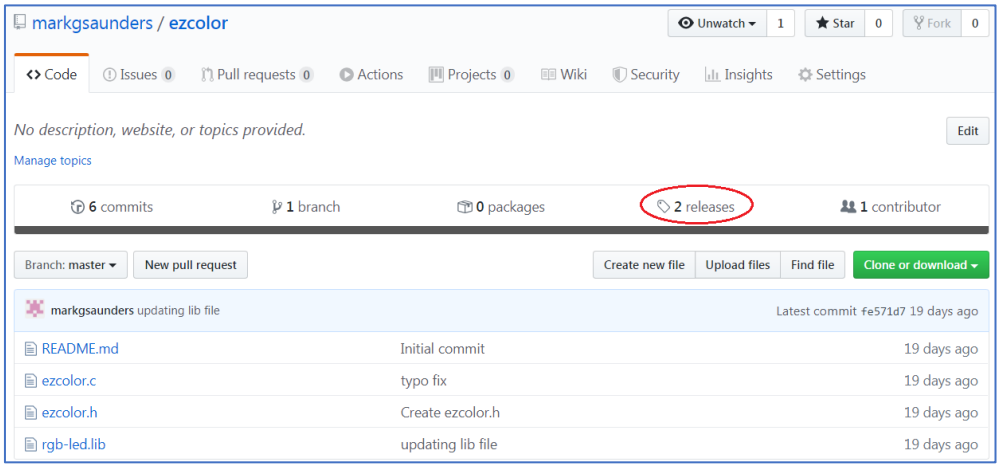
```
#include "ezcolor.h"
cy_rslt_t init_colors( void )
{
  return cy_rgb_led_init( CYBSP_LED_RGB_RED, CYBSP_LED_RGB_GREEN,
CYBSP_LED_RGB_BLUE, CY_RGB_LED_ACTIVE_LOW );
}
void color( color_t color )
{
  cy_rgb_led_on( color, CY_RGB_LED_MAX_BRIGHTNESS );
}
```

10. Repeat the process a final time to create “rgb-led.lib” with the following content:

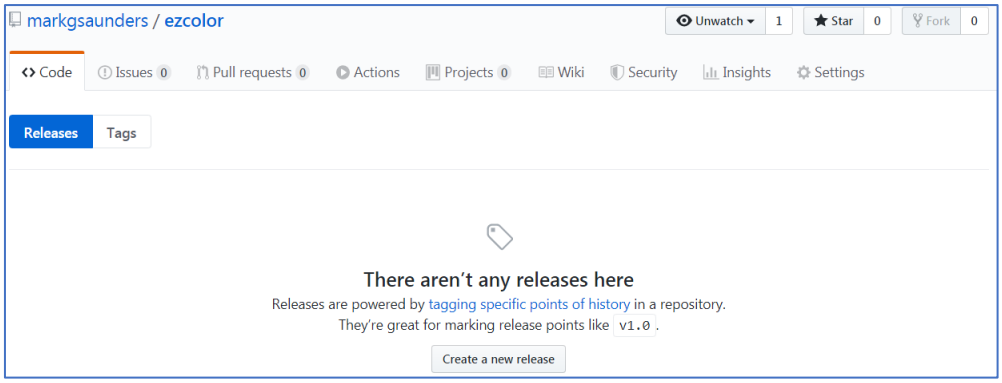
    `https://github.com/cypresssemiconductorco/rgb-led/#latest-v1.X`

    This lib file will cause the `make getlibs` command to automatically pull in the required rgb-led library along with ez-color.

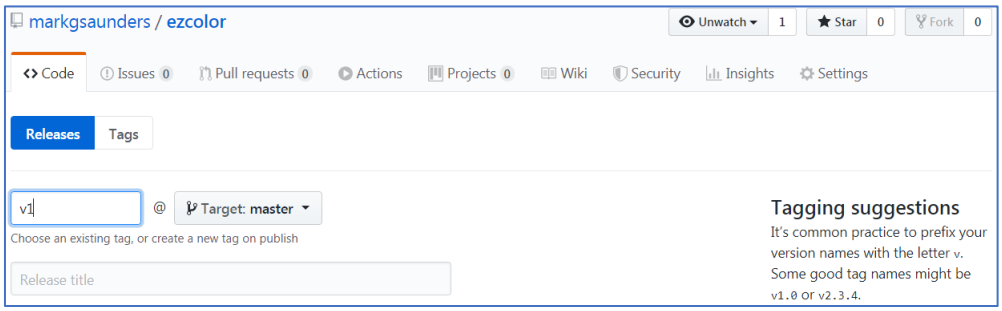
11. Your repo should now contain four files; the readme, a header, a source file, and the lib file.



12. Next your need to tag your repo in a release so click on the Releases tab as shown.



13. Press the Create a new release button. Give your release a version string and Publish it - I called mine "v1".



14. Your library is now ready to use. Open a Modus Shell by running this batch file.

```
~/ModusToolbox/tools_2.1/modus-shell/Cygwin.bat
```

15. In the shell create and cd into a suitable directory for your project.

16. Create a new project by cloning the Cypress empty app example.

```
git clone https://github.com/cypresssemiconductorco/mtb-example-psoc6-empty-app
Cloning into 'mtb-example-psoc6-empty-app'...
remote: Enumerating objects: 13, done.
remote: Counting objects: 100% (13/13), done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 13 (delta 0), reused 10 (delta 0), pack-reused 0
Unpacking objects: 100% (13/13), done.
```

17. Rename the directory to **ch05a_ex11_new_library** and change into the project directory.

```
mv mtb-example-psoc6-empty-app ch05a_ex11_new_library
cd ch05a_ex11_library
```

18. Create a .lib file to tell ModusToolbox to include your new library in this project. The file will contain a single line:

```
https://github.com/<YOUR_GITHUB_NAME>/<YOUR_REPO_NAME>/#<YOUR_RELEASE_TAG>
```

Be very careful with the punctuation, particularly the "/#" between the repo and tag names. You can create this file with an editor or a command like (but not the same as) this:

```
echo "https://github.com/markgsaunders/ezcolor/#v1" > ezcolor.lib
cat ezcolor.lib
https://github.com/markgsaunders/ezcolor/#v1
```

19. Now run `make getlibs` to pull in all the library firmware into your project.

```
make getlibs
Tools Directory: C:/Users/yfs/ModusToolbox/tools_2.1
Initializing import: ch05a_ex11_new_library
================================================================================
= Importing libraries =
================================================================================
Git is git version 2.17.0, found at /usr/bin/git
Searching application directories...
Found 1 file(s)
    Processing file C:/Users/yfs/colortest/ch05a_ex11_new_library/ezcolor.lib
Application directories search complete.
Searching libs directory...
Found 8 file(s)
    Processing file
C:/Users/yfs/colortest/ch05a_ex11_new_library/libs/TARGET_CY8CPROTO-062-4343W.lib
    Processing file C:/Users/yfs/colortest/ch05a_ex11_new_library/libs/ezcolor/rgb-
led.lib
    Processing file
C:/Users/yfs/colortest/ch05a_ex11_new_library/libs/TARGET_CY8CPROTO-062-
4343W/libs/capsense.lib
    Processing file
C:/Users/yfs/colortest/ch05a_ex11_new_library/libs/TARGET_CY8CPROTO-062-
4343W/libs/core-lib.lib
    Processing file
C:/Users/yfs/colortest/ch05a_ex11_new_library/libs/TARGET_CY8CPROTO-062-
4343W/libs/psoc6cm0p.lib
    Processing file
C:/Users/yfs/colortest/ch05a_ex11_new_library/libs/TARGET_CY8CPROTO-062-
4343W/libs/psoc6hal.lib
    Processing file
C:/Users/yfs/colortest/ch05a_ex11_new_library/libs/TARGET_CY8CPROTO-062-
4343W/libs/psoc6make.lib
    Processing file
C:/Users/yfs/colortest/ch05a_ex11_new_library/libs/TARGET_CY8CPROTO-062-
4343W/libs/psoc6pdl.lib
Libraries were processed. Re-evaluating libs directory...
Found 8 file(s)
libs directory search complete.
================================================================================
= Import complete =
================================================================================
```

20. Unfortunately, this example does not include the BSP for the CY8CKIT-062S2-43012 kit, so we need to add it. It is a simple git command to fetch the required firmware.

```
git clone https://github.com/cypresssemiconductorco/TARGET_CY8CKIT-062S2-43012
Cloning into 'TARGET_CY8CKIT-062S2-43012'...
remote: Enumerating objects: 254, done.
remote: Counting objects: 100% (254/254), done.
remote: Compressing objects: 100% (155/155), done.
remote: Total 254 (delta 98), reused 244 (delta 91), pack-reused 0
Receiving objects: 100% (254/254), 1.08 MiB | 1.99 MiB/s, done.
Resolving deltas: 100% (98/98), done.
Checking out files: 100% (221/221), done.
```

21. Edit your Makefile to change the target BSP.

```
############################
# Basic Configuration
############################
# Target board/hardware
# TARGET=CY8CPROTO-062-4343W
TARGET= CY8CKIT-062S2-43012
```

22. Overwrite your main.c with the following code.

```
#include "cy_pdl.h"
#include "cyhal.h"
#include "cybsp.h"
#include "ezcolor.h"
int main(void)
{
    cy_rslt_t result;
    /* Initialize the device and board peripherals */
    result = cybsp_init() ;
    if (result != CY_RSLT_SUCCESS)
    {
        CY_ASSERT(0);
    }
    __enable_irq();
    init_colors();
    for(;;)
    {
        color( RED );
        Cy_SysLib_Delay( 1000 );
        color( GREEN );
        Cy_SysLib_Delay( 1000 );
        color( BLUE );
        Cy_SysLib_Delay( 1000 );
        color( YELLOW );
        Cy_SysLib_Delay( 1000 );
        color( CYAN );
        Cy_SysLib_Delay( 1000 );
        color( MAGENTA );
        Cy_SysLib_Delay( 1000 );
        color( WHITE );
        Cy_SysLib_Delay( 1000 );
        color( BLACK );
        Cy_SysLib_Delay( 1000 );
    }
}
```

23. Run `make program` and observe the LED colors.

5a.17.10 Exercise 12: Create and Use a New BSP

1. From the command line, create a new project: Hello World. Rename it to **ch05a_ex12_new_bsp** and change to that directory.

```
git clone https://github.com/cypresssemiconductorco/mtb-example-psoc6-Hello-World
mv mtb-example-psoc6-Hello-World ch05a_ex12_new_bsp
cd ch05a_ex12_new_bsp
```

2. Pull in the BSP for your kit. (This would be done for you by the Project Creator tool if you had used it instead of manually cloning.)

```
git clone https://github.com/cypresssemiconductorco/TARGET_CY8CKIT-062S2-43012
```

3. Edit the TARGET variable in the Makefile to use your kit.

   (This would be done for you by the Project Creator tool if you had used it instead of manually cloning.)

   ```
   TARGET= CY8CKIT-062S2-43012
   ```

4. Run `make getlibs` to pull in all the library firmware into your project.

   ```
   make getlibs
   ```

5. Build and program the kit and observe the LED that pauses/resumes blinking when you press the return key in your terminal emulator.

   ```
   make program
   ```

6. Create a new BSP called “MYKIT”

   ```
   make bsp TARGET_GEN=MYKIT
   ```

7. In the new directory (TARGET_MYKIT), edit the cybsp_types.h file to swap the LEDs assigned to CYBSP_USER_LED1 and CYBSP_USER_LED2.

   ```
   /** LED 9; User LED1 */
   #define CYBSP_USER_LED1             (CYBSP_LED9)
   /** LED 8; User LED2 */
   #define CYBSP_USER_LED2             (CYBSP_LED8)
   ```

8. Build and program the kit with your new BSP and observe a different LED blinking.

   ```
   make program TARGET=MYKIT
   ```