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 supported kits
- Peripheral Driver Library (PDL)
- Libraries (Retarget-IO, emWin etc.)
- Programming and debug tools

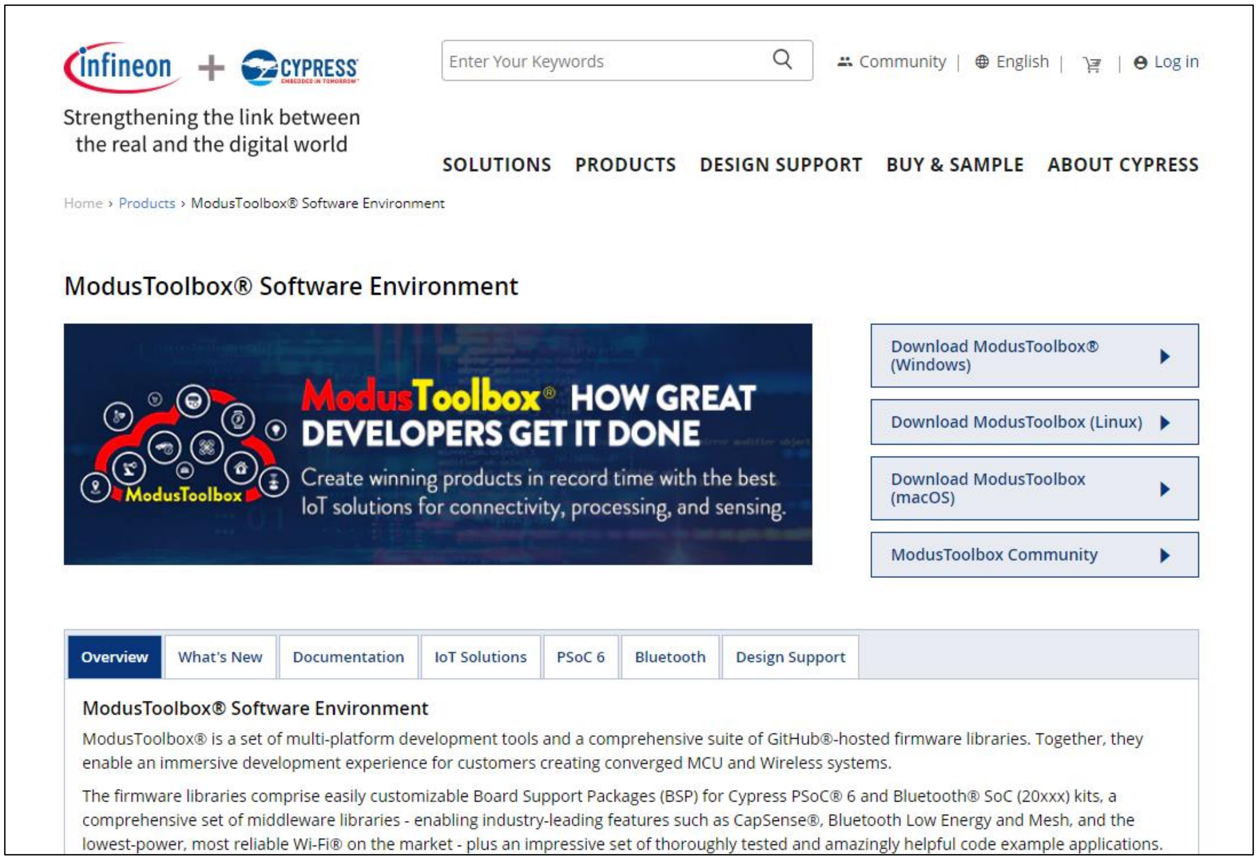
This section provides a brief tour of the ModusToolbox websites, Eclipse IDE, and various tools.

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


Strengthening the link between the real and the digital world
Enter Your Keywords
Community | English | | Log in
SOLUTIONS
PRODUCTS
DESIGN SUPPORT
BUY & SAMPLE
ABOUT CYPRESS
Home › Products › ModusToolbox® Software Environment
ModusToolbox® Software Environment
ModusToolbox® HOW GREAT DEVELOPERS GET IT DONE
Create winning products in record time with the best IoT solutions for connectivity, processing, and sensing.
Download ModusToolbox® (Windows)
Download ModusToolbox (Linux)
Download ModusToolbox (macOS)
ModusToolbox Community
Overview
What's New
Documentation
IoT Solutions
PSoC 6
Bluetooth
Design Support

ModusToolbox® Software Environment

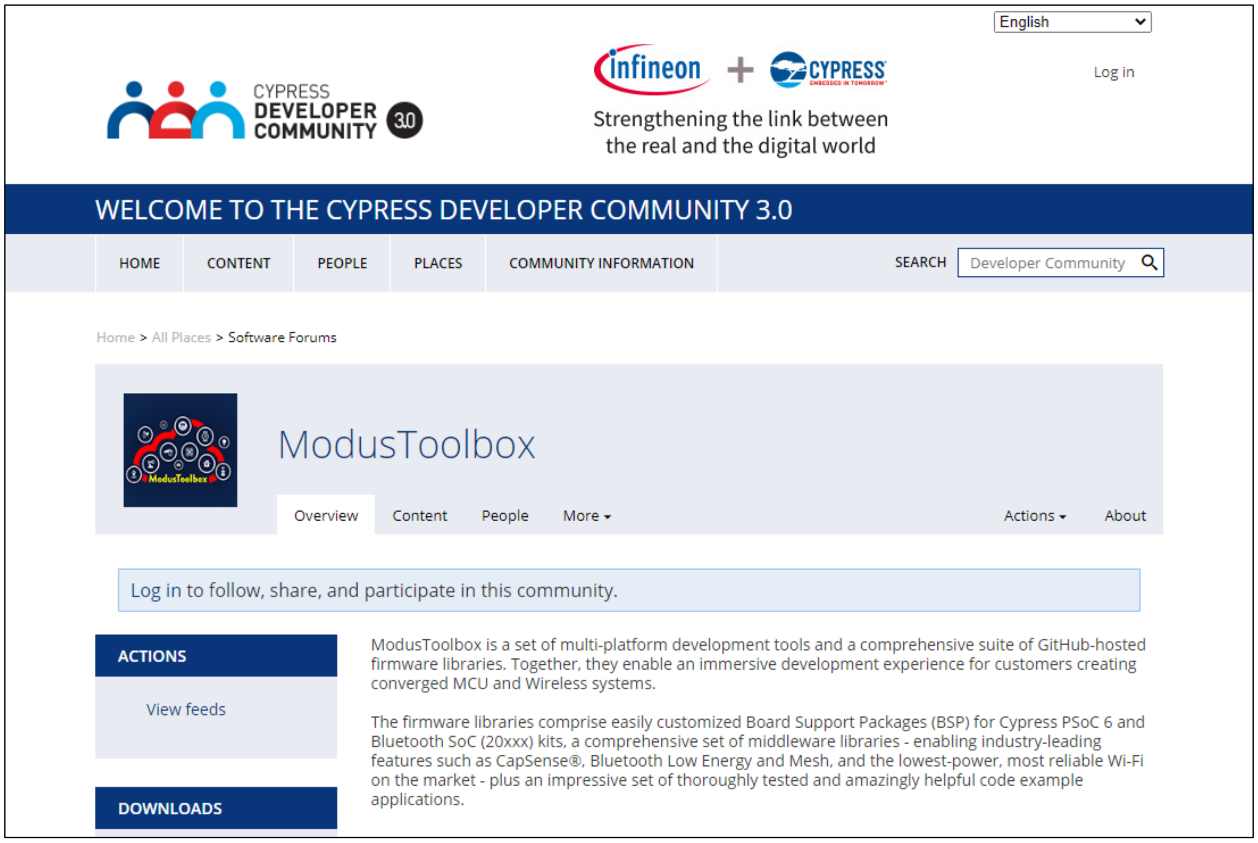
ModusToolbox® is a set of multi-platform development tools and a comprehensive suite of GitHub®-hosted firmware libraries. Together, they enable an immersive development experience for customers creating converged MCU and Wireless systems.

The firmware libraries comprise easily customizable Board Support Packages (BSP) for Cypress PSoC® 6 and Bluetooth® SoC (20xxx) kits, a comprehensive set of middleware libraries - enabling industry-leading features such as CapSense®, Bluetooth Low Energy and Mesh, and the lowest-power, most reliable Wi-Fi® on the market - plus an impressive set of thoroughly tested and amazingly helpful code example applications.

Community

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

https://community.cypress.com/community/software-forums/modustoolbox


English
Log in
CYPRESS DEVELOPER COMMUNITY 3.0
Strengthening the link between the real and the digital world
WELCOME TO THE CYPRESS DEVELOPER COMMUNITY 3.0
HOME
CONTENT
PEOPLE
PLACES
COMMUNITY INFORMATION
SEARCH
Developer Community
Home > All Places > Software Forums
ModusToolbox
Overview
Content
People
More
Actions
About
ACTIONS
View feeds
DOWNLOADS

Log in to follow, share, and participate in this community.

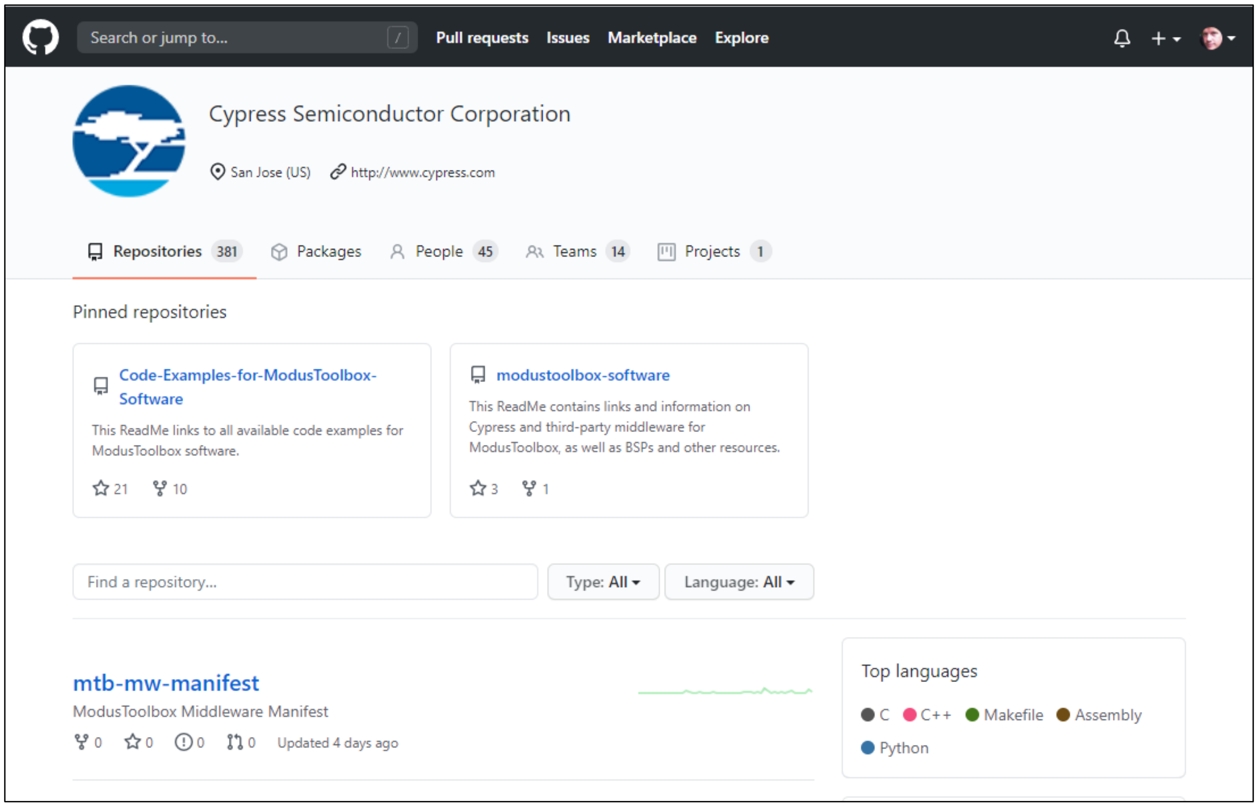
ModusToolbox is a set of multi-platform development tools and a comprehensive suite of GitHub-hosted firmware libraries. Together, they enable an immersive development experience for customers creating converged MCU and Wireless systems.

The firmware libraries comprise easily customized Board Support Packages (BSP) for Cypress PSoC 6 and Bluetooth SoC (20xxx) kits, a comprehensive set of middleware libraries - enabling industry-leading features such as CapSense®, Bluetooth Low Energy and Mesh, and the lowest-power, most reliable Wi-Fi on the market - plus an impressive set of thoroughly tested and amazingly helpful code example applications.

Github.com

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

https://github.com/cypresssemiconductorco



Cypress Semiconductor Corporation

San Jose (US) http://www.cypress.com

Pinned repositories

Code-Examples-for-ModusToolbox-Software
This ReadMe links to all available code examples for ModusToolbox software.
21 10

modustoolbox-software
This ReadMe contains links and information on Cypress and third-party middleware for ModusToolbox, as well as BSPs and other resources.
3 1

mtb-mw-manifest
ModusToolbox Middleware Manifest
0 0 0 0 Updated 4 days ago

Top languages
● C ● C++ ● Makefile ● Assembly
● Python

Note If you want to see HTML documentation from GitHub, you can access it using `cypresssemiconductorco.github.io` so that it is rendered properly. For example, if you go to the GitHub site for the RGB LED library, you will see this path to the documentation:

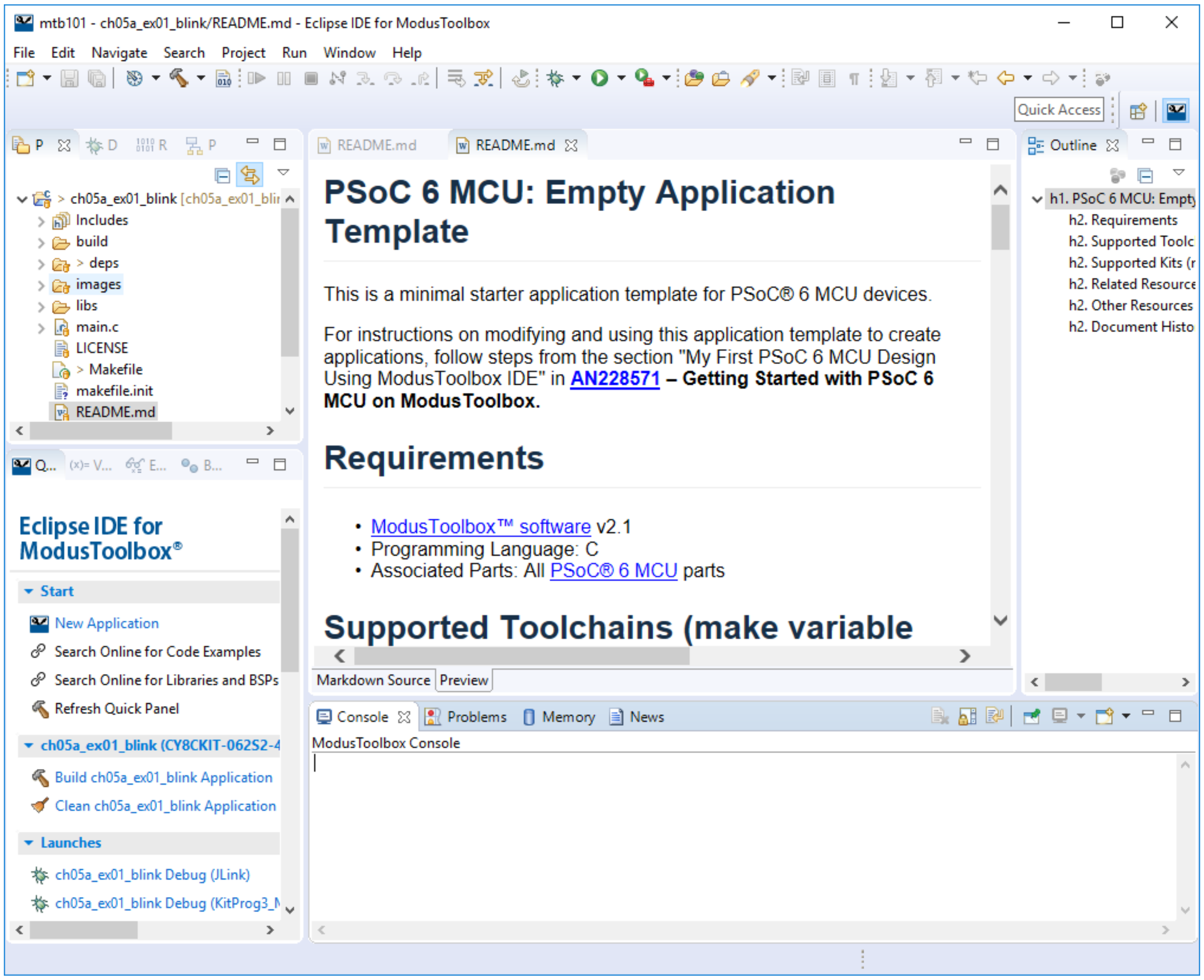
https://github.com/cypresssemiconductorco/rgb-led/blob/master/docs/api_reference_manual.html

To see the documentation properly rendered, change the URL to the following:

https://cypresssemiconductorco.github.io/rgb-led/api_reference_manual.html

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.


mtb101 - ch05a_ex01_blink/README.md - Eclipse IDE for ModusToolbox
File Edit Navigate Search Project Run Window Help
Quick Access
ch05a_ex01_blink [ch05a_ex01_blin
Includes
build
deps
images
libs
main.c
LICENSE
Makefile
makefile.init
README.md
Eclipse IDE for ModusToolbox®
Start
New Application
Search Online for Code Examples
Search Online for Libraries and BSPs
Refresh Quick Panel
ch05a_ex01_blink (CY8CKIT-062S2-4
Build ch05a_ex01_blink Application
Clean ch05a_ex01_blink Application
Launches
ch05a_ex01_blink Debug (JLink)
ch05a_ex01_blink Debug (KitProg3_M
README.md
Markdown Source
Preview
Outline
h1. PSoC 6 MCU: Empty
h2. Requirements
h2. Supported Toolc
h2. Supported Kits (r
h2. Related Resource
h2. Other Resources
h2. Document Histo
Console
Problems
Memory
News
ModusToolbox Console

PSoC 6 MCU: Empty Application Template

This is a minimal starter application template for PSoC® 6 MCU devices.

For instructions on modifying and using this application template to create applications, follow steps from the section "My First PSoC 6 MCU Design Using ModusToolbox IDE" in AN228571 – **Getting Started with PSoC 6 MCU on ModusToolbox.**

Requirements

- ModusToolbox™ software v2.1
- Programming Language: C
- Associated Parts: All PSoC® 6 MCU parts

Supported Toolchains (make variable

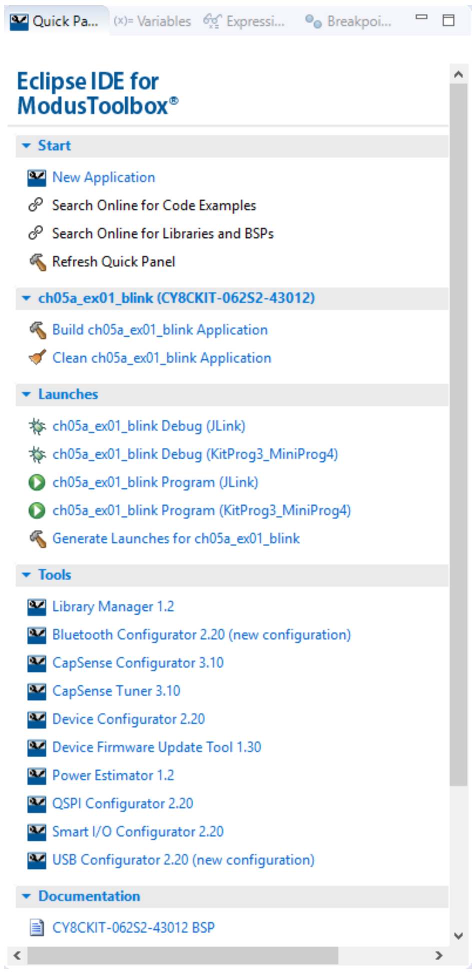
For more information about the IDE, refer to the “Eclipse IDE for ModusToolbox User Guide”:

http://www.cypress.com/MTBEclipseIDEUserGuide

Quick Panel

We have 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


Quick Pa... (x)= Variables Expressi... Breakpoi...

Eclipse IDE for ModusToolbox®

Start
- New Application
- Search Online for Code Examples
- Search Online for Libraries and BSPs
- Refresh Quick Panel

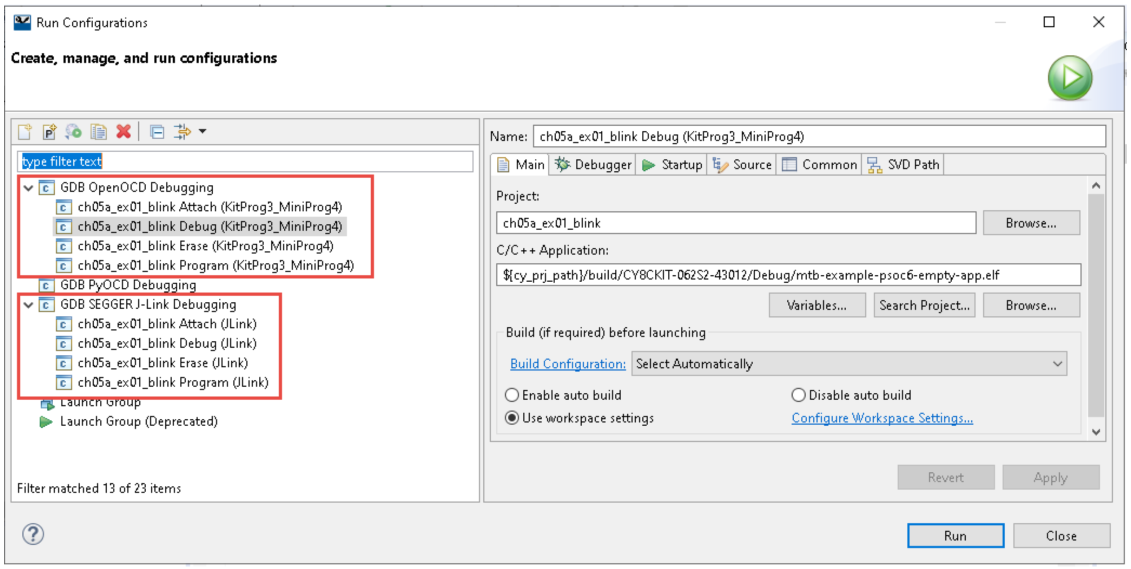
ch05a_ex01_blink (CY8CKIT-062S2-43012)
- Build ch05a_ex01_blink Application
- Clean ch05a_ex01_blink Application

Launches
- ch05a_ex01_blink Debug (JLink)
- ch05a_ex01_blink Debug (KitProg3_MiniProg4)
- ch05a_ex01_blink Program (JLink)
- ch05a_ex01_blink Program (KitProg3_MiniProg4)
- Generate Launches for ch05a_ex01_blink

Tools
- Library Manager 1.2
- Bluetooth Configurator 2.20 (new configuration)
- CapSense Configurator 3.10
- CapSense Tuner 3.10
- Device Configurator 2.20
- Device Firmware Update Tool 1.30
- Power Estimator 1.2
- QSPI Configurator 2.20
- Smart I/O Configurator 2.20
- USB Configurator 2.20 (new configuration)

Documentation
- CY8CKIT-062S2-43012 BSP

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** and **GDB SEGGER J-Link 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.


Run Configurations

Create, manage, and run configurations

type filter text

- GDB OpenOCD Debugging
  - ch05a_ex01_blink Attach (KitProg3_MiniProg4)
  - ch05a_ex01_blink Debug (KitProg3_MiniProg4)
  - ch05a_ex01_blink Erase (KitProg3_MiniProg4)
  - ch05a_ex01_blink Program (KitProg3_MiniProg4)
- GDB PyOCD Debugging
- GDB SEGGER J-Link Debugging
  - ch05a_ex01_blink Attach (JLink)
  - ch05a_ex01_blink Debug (JLink)
  - ch05a_ex01_blink Erase (JLink)
  - ch05a_ex01_blink Program (JLink)
- Launch Group
- Launch Group (Deprecated)

Filter matched 13 of 23 items

Name: ch05a_ex01_blink Debug (KitProg3_MiniProg4)

Main | Debugger | Startup | Source | Common | SVD Path

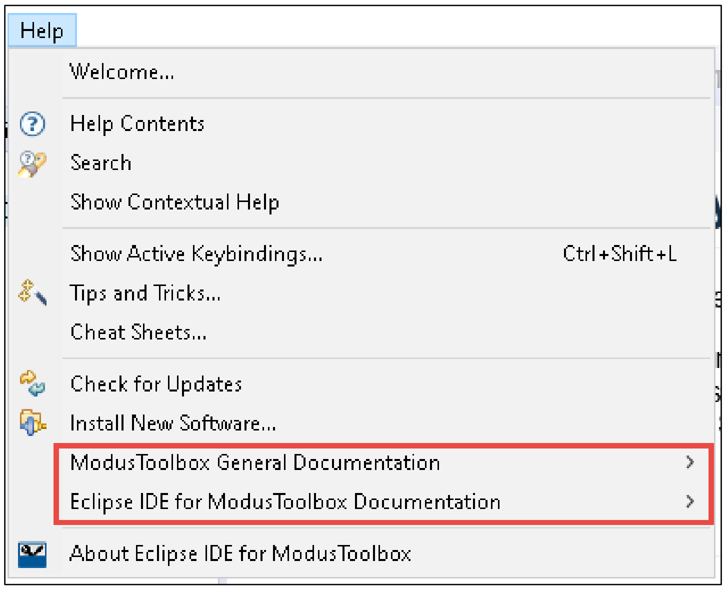
Project:
ch05a_ex01_blink

C/C++ Application:
${cy_prj_path}/build/CY8CKIT-062S2-43012/Debug/mtb-example-psoc6-empty-app.elf

Build (if required) before launching
Build Configuration: Select Automatically
Enable auto build
Disable auto build
Use workspace settings
Configure Workspace Settings...

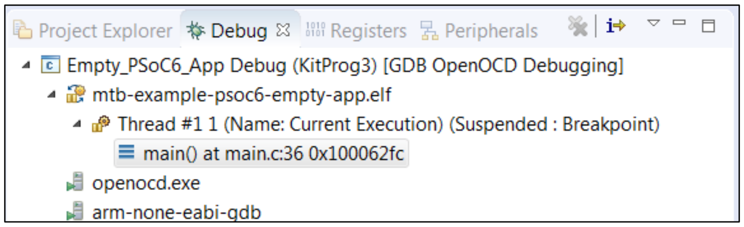
Help Menu

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.


Help
Welcome...
Help Contents
Search
Show Contextual Help
Show Active Keybindings... Ctrl+Shift+L
Tips and Tricks...
Cheat Sheets...
Check for Updates
Install New Software...
ModusToolbox General Documentation
Eclipse IDE for ModusToolbox Documentation
About Eclipse IDE for ModusToolbox

Integrated Debugger

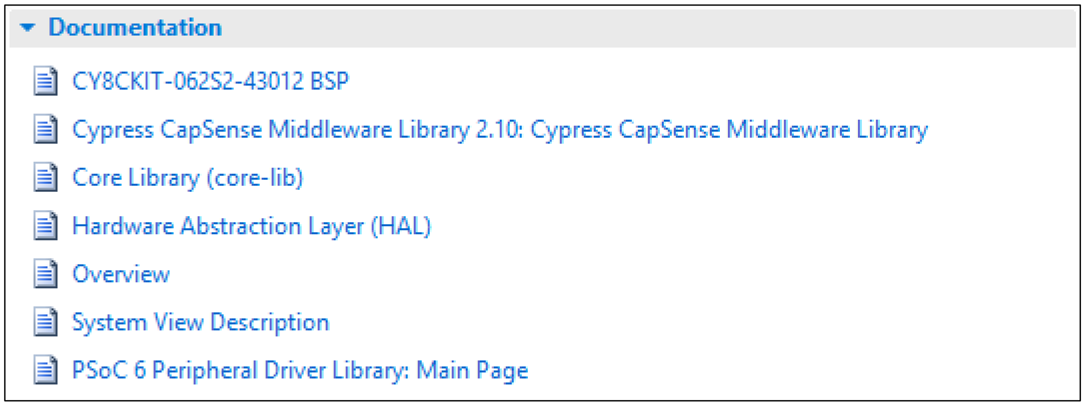
The Eclipse IDE provides an integrated debugger using either the KitProg3 (OpenOCD) on the PSoC 6 development kit, or through a MiniProg4 or Segger J-Link debug probe.



Empty_PSoC6_App Debug (KitProg3) [GDB OpenOCD Debugging]
mtb-example-psoc6-empty-app.elf
Thread #1 1 (Name: Current Execution) (Suspended : Breakpoint)
main() at main.c:36 0x100062fc
openocd.exe
arm-none-eabi-gdb

Documentation

After creating a project, assorted links to documentation are available directly from the Quick Panel, under the "Documentation" section. This will update to include documentation for new libraries as you add them to the application.



Documentation

- CY8CKIT-062S2-43012 BSP
- Cypress CapSense Middleware Library 2.10: Cypress CapSense Middleware Library
- Core Library (core-lib)
- Hardware Abstraction Layer (HAL)
- Overview
- System View Description
- PSoC 6 Peripheral Driver Library: Main Page

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:

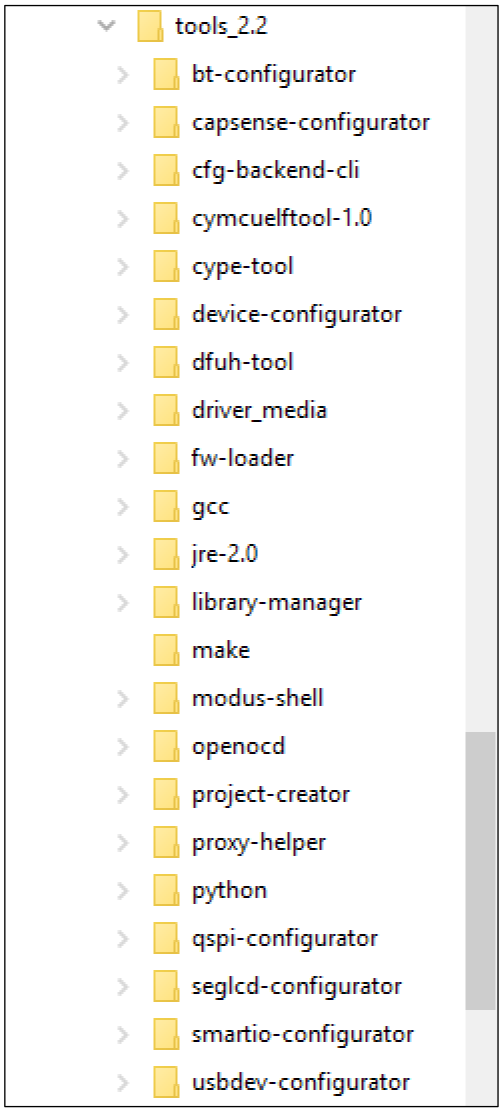
- If your code has IntelliSense issues, use **Program > Rebuild Index** and/or **Program > Build**.
- Sometimes when you import a project, you don't see all the files. Right-click on the project and select **Refresh**.
- Various menus and the Quick Panel show different things depending on what you select 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 our 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_2.2:



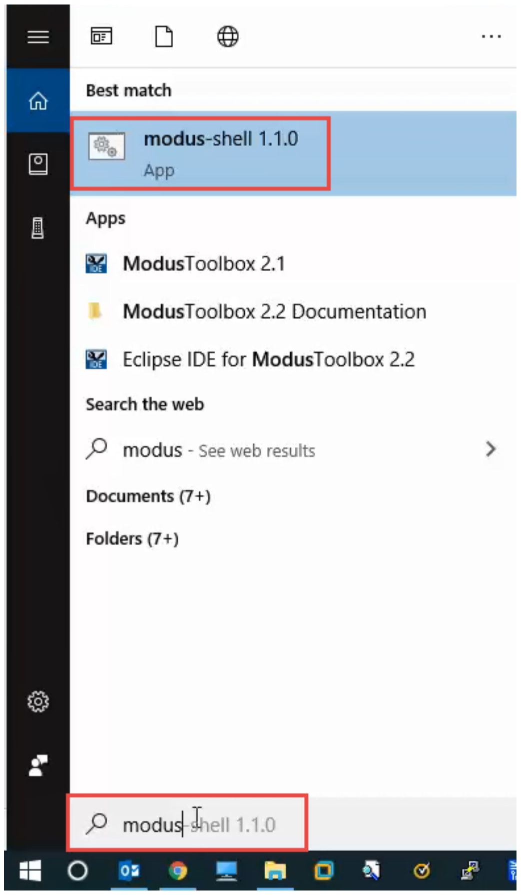
```
tools_2.2
    bt-configurator
    capsense-configurator
    cfg-backend-cli
    cymcuelftool-1.0
    cype-tool
    device-configurator
    dfuh-tool
    driver_media
    fw-loader
    gcc
    jre-2.0
    library-manager
    make
    modus-shell
    openocd
    project-creator
    proxy-helper
    python
    qspi-configurator
    seglcd-configurator
    smartio-configurator
    usbdev-configurator
```

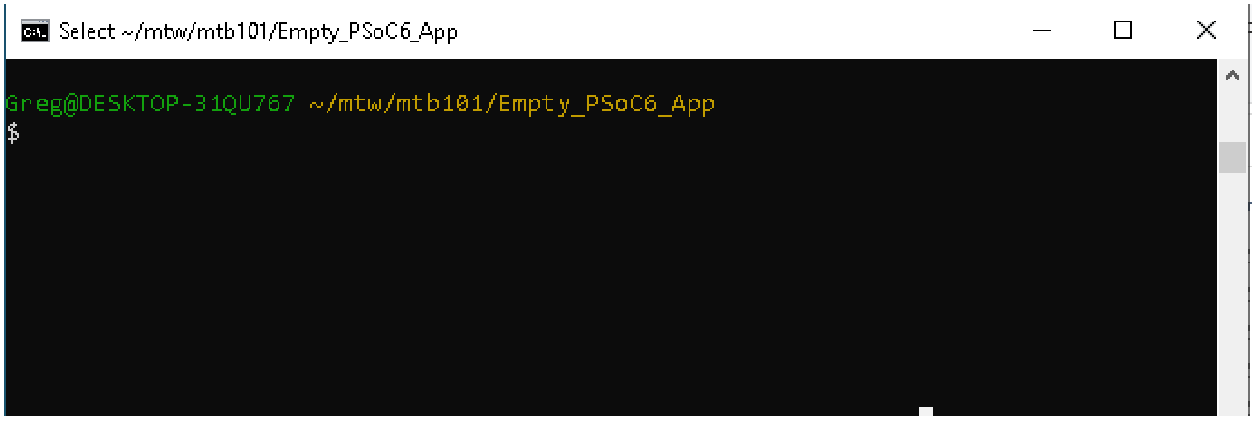
5a.1.4 Command Line

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 modus-shell (it is in the ModusToolbox installation under *ModusToolbox/tools_2.2/modus-shell*). It is also listed in the Start menu under ModusToolbox 2.2.




Select ~/mtw/mtb101/Empty_PSoC6_App

```
Greg@DESKTOP-31QU767 ~/mtw/mtb101/Empty_PSoC6_App
$
```

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 a few helpful make targets (i.e. commands). Refer also to the ModusToolbox User Guide document.

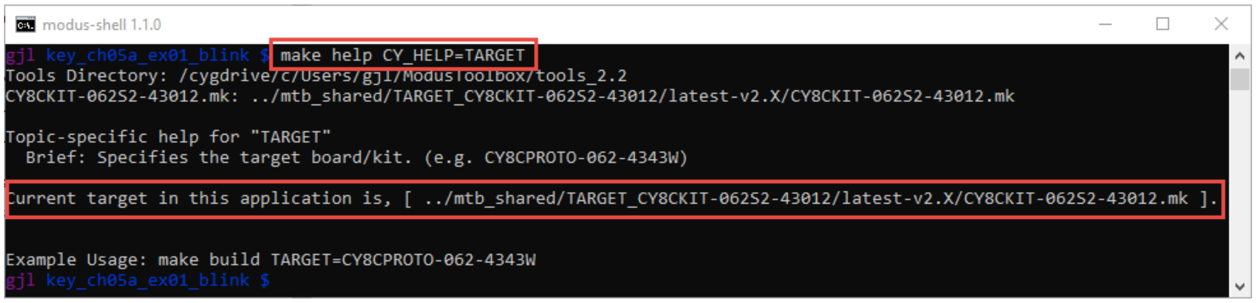
Make Command	**Description**
`make help`	This command will print out a list of all the make targets. To get help on a specific target type `make help CY_HELP=getlibs` (or whichever target you want help with).
`make getlibs`	Process all the *mtb* files (or *lib* files) and bring all the libraries into 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.
`make get_app_info`	Prints all variable settings for the app.
`make get_env_info`	Prints the tool versions that are being used.
`make printlibs`	Prints information about all the libraries including Git versions.

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
```



```
gjl key_ch05a_ex01_blink $ make help CY_HELP=TARGET
Tools Directory: /cygdrive/c/Users/gjl/ModusToolbox/tools_2.2
CY8CKIT-062S2-43012.mk: ../mtb_shared/TARGET_CY8CKIT-062S2-43012/latest-v2.X/CY8CKIT-062S2-43012.mk

Topic-specific help for "TARGET"
  Brief: Specifies the target board/kit. (e.g. CY8CPROTO-062-4343W)

Current target in this application is, [ ../mtb_shared/TARGET_CY8CKIT-062S2-43012/latest-v2.X/CY8CKIT-062S2-43012.mk ].


Example Usage: make build TARGET=CY8CPROTO-062-4343W
gjl key_ch05a_ex01_blink $
```

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 modlibs` launches the Library Manager and opens the application's library settings.

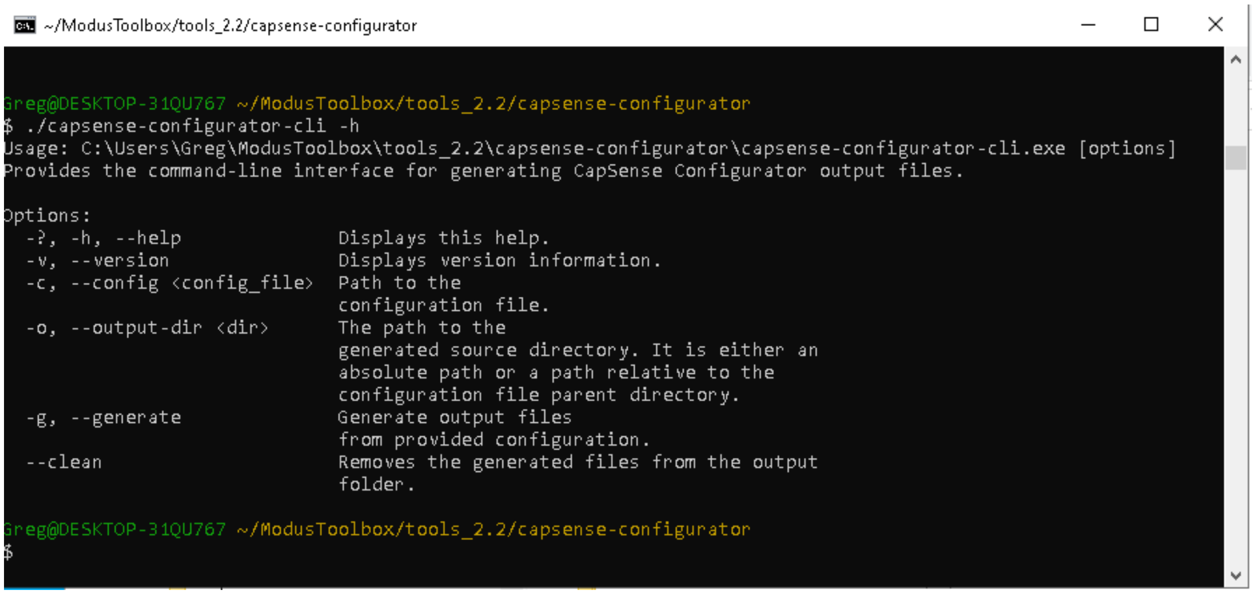
You can launch other ModusToolbox tools using `make open` and specifying a tool name or a file to open. Use `make help CY_HELP=open` for details. For example:

- `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_2.2. 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
```



~/ModusToolbox/tools_2.2/capsense-configurator

```
Greg@DESKTOP-31QU767 ~/ModusToolbox/tools_2.2/capsense-configurator
$ ./capsense-configurator-cli -h
Usage: C:\Users\Greg\ModusToolbox\tools_2.2\capsense-configurator\capsense-configurator-cli.exe [options]
Provides the command-line interface for generating CapSense Configurator output files.

Options:
  -?, -h, --help              Displays this help.
  -v, --version               Displays version information.
  -c, --config <config_file>  Path to the
                              configuration file.
  -o, --output-dir <dir>      The path to the
                              generated source directory. It is either an
                              absolute path or a path relative to the
                              configuration file parent directory.
  -g, --generate              Generate output files
                              from provided configuration.
  --clean                     Removes the generated files from the output
                              folder.

Greg@DESKTOP-31QU767 ~/ModusToolbox/tools_2.2/capsense-configurator
$
```

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

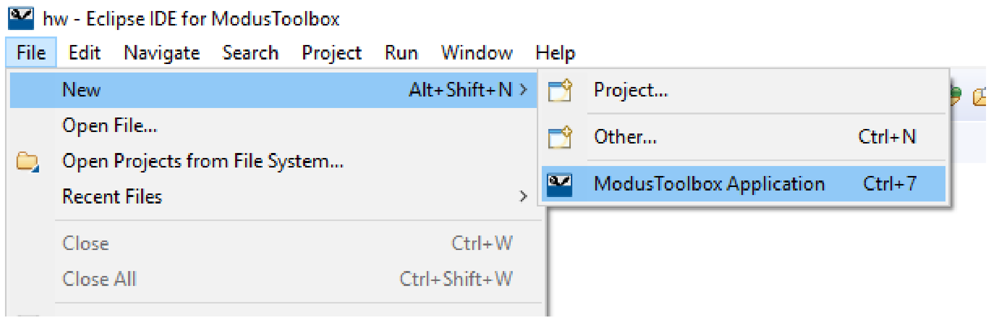
Repeat demonstration but using the command-line tools rather than an IDE.

5a.3 Exercises (Part 1)

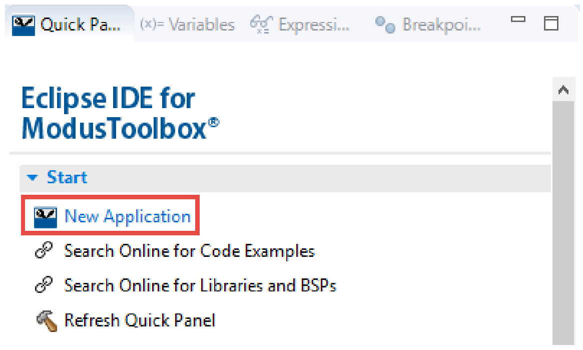
5a.3.1 Exercise 1: Blinking an LED from the Eclipse IDE

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 *mtb101* 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**.

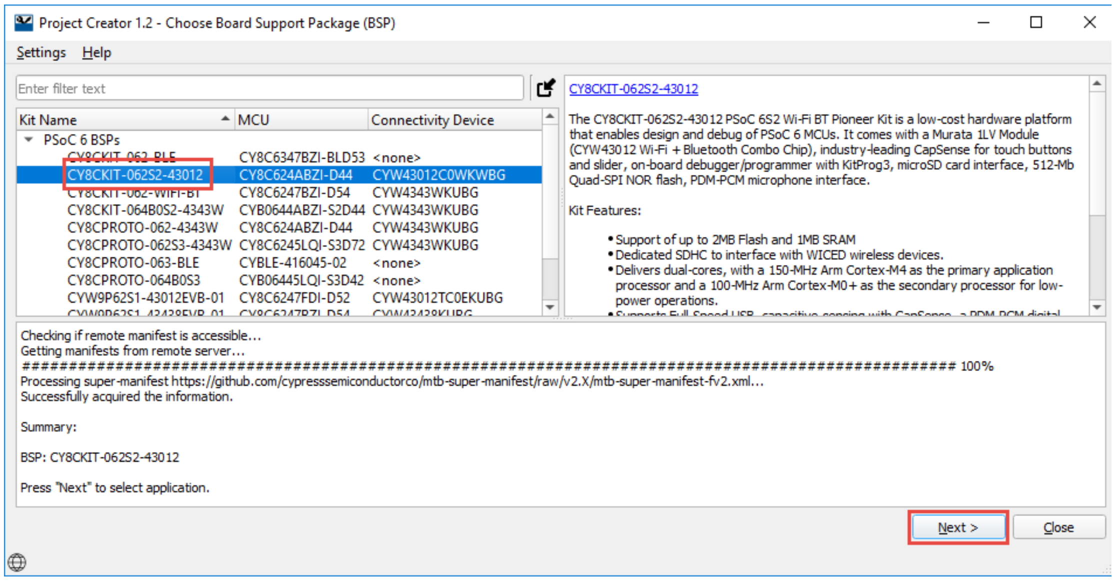


Or on the Quick Panel, click the **New Application** link.


Quick Pa...
(x)= Variables
Expressi...
Breakpoi...
Eclipse IDE for ModusToolbox®
Start
New Application
Search Online for Code Examples
Search Online for Libraries and BSPs
Refresh Quick Panel

Either option launches the ModusToolbox Project Creator tool.

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



Project Creator 1.2 - Choose Board Support Package (BSP)

Settings Help

Enter filter text

Kit Name	MCU	Connectivity Device
PSoC 6 BSPs		
CY8CKIT-062-BLE	CY8C6347BZI-BLD53	<none>
CY8CKIT-062S2-43012	CY8C624ABZI-D44	CYW43012C0WKWBG
CY8CKIT-062-WIFI-BT	CY8C6247BZI-D54	CYW4343WKUBG
CY8CKIT-064B0S2-4343W	CYB0644ABZI-S2D44	CYW4343WKUBG
CY8CPROTO-062-4343W	CY8C624ABZI-D44	CYW4343WKUBG
CY8CPROTO-062S3-4343W	CY8C6245LQI-S3D72	CYW4343WKUBG
CY8CPROTO-063-BLE	CYBLE-416045-02	<none>
CY8CPROTO-064B0S3	CYB06445LQI-S3D42	<none>
CYW9P62S1-43012EVB-01	CY8C6247FDI-D52	CYW43012TC0EKUBG

CY8CKIT-062S2-43012

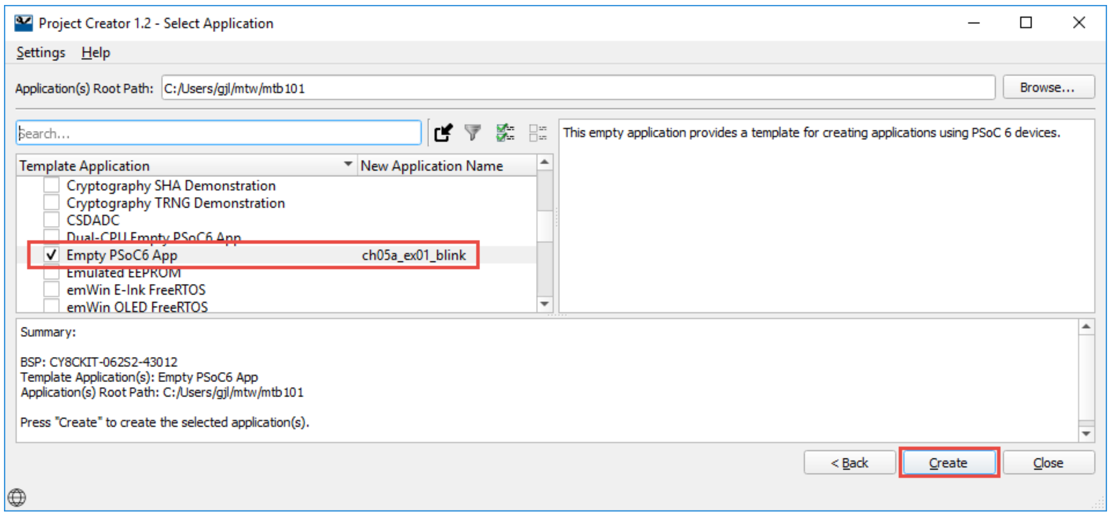
The CY8CKIT-062S2-43012 PSoC 6S2 Wi-Fi BT Pioneer Kit is a low-cost hardware platform that enables design and debug of PSoC 6 MCUs. It comes with a Murata 1LV Module (CYW43012 Wi-Fi + Bluetooth Combo Chip), industry-leading CapSense for touch buttons and slider, on-board debugger/programmer with KitProg3, microSD card interface, 512-Mb Quad-SPI NOR flash, PDM-PCM microphone interface.

Kit Features:

- Support of up to 2MB Flash and 1MB SRAM
- Dedicated SDHC to interface with WICED wireless devices.
- Delivers dual-cores, with a 150-MHz Arm Cortex-M4 as the primary application processor and a 100-MHz Arm Cortex-M0+ as the secondary processor for low-power operations.

```
Checking if remote manifest is accessible...
Getting manifests from remote server...
############################################################################################## 100%
Processing super-manifest https://github.com/cypresssemiconductorco/mtb-super-manifest/raw/v2.X/mtb-super-manifest-fv2.xml...
Successfully acquired the information.

Summary:

BSP: CY8CKIT-062S2-43012

Press "Next" to select application.
```

☐ 3. Check the box next to *Empty PSoC 6 App* and give your project a name (in this case **ch05a_ex01_blink**). Then, click **Create**.



Project Creator 1.2 - Select Application

Settings Help

Application(s) Root Path: C:/Users/gjl/mtw/mtb101

Search...

Template Application	New Application Name
☐ Cryptography SHA Demonstration	
☐ Cryptography TRNG Demonstration	
☐ CSDADC	
☐ Dual-CPU Empty PSoC6 App	
✓ Empty PSoC6 App	ch05a_ex01_blink
☐ Emulated EEPROM	
☐ emWin E-Ink FreeRTOS	
☐ emWin OLED FreeRTOS	

This empty application provides a template for creating applications using PSoC 6 devices.

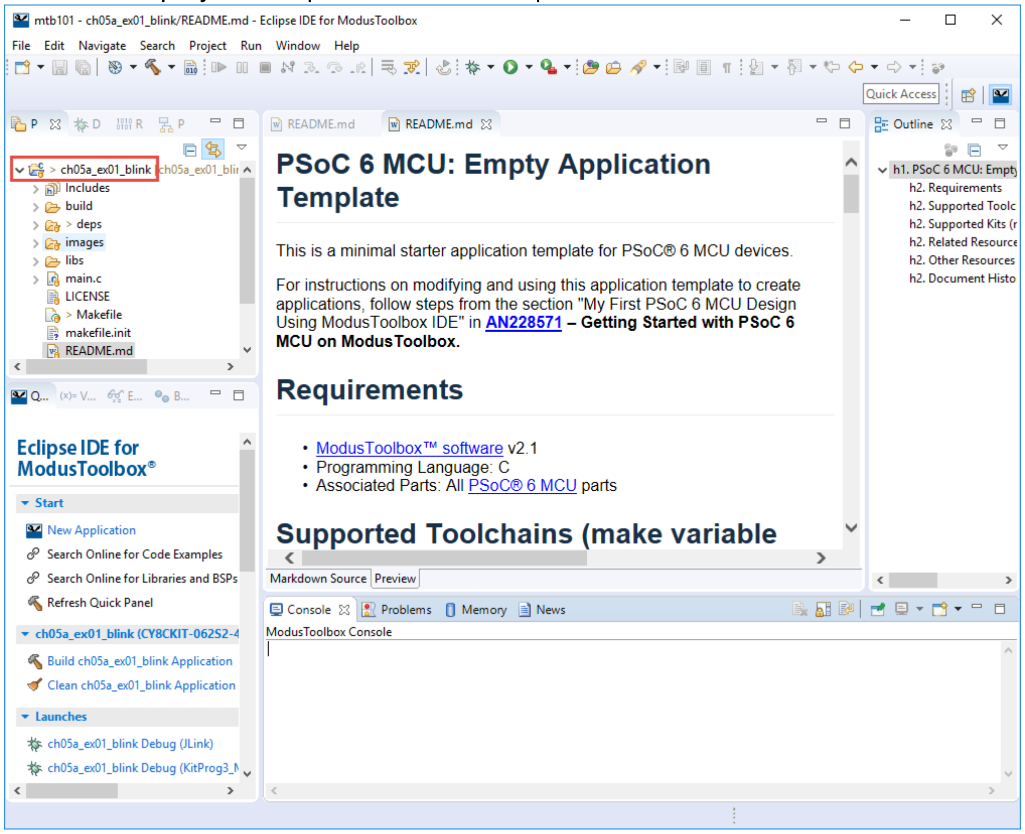
```
Summary:

BSP: CY8CKIT-062S2-43012
Template Application(s): Empty PSoC6 App
Application(s) Root Path: C:/Users/gjl/mtw/mtb101

Press "Create" to create the selected application(s).
```

You can check the box for more than one application at a time if you want to create multiple applications in a single operation.

There are application searching and filtering capabilities which we will talk about in the Project Creator Tool section.

☐ 4. At this point, the application is created.

Once creation is done, the new project creation window will close (unless there are errors) and the new project is imported into the Eclipse IDE.



mtb101 - ch05a_ex01_blink/README.md - Eclipse IDE for ModusToolbox

File Edit Navigate Search Project Run Window Help

ch05a_ex01_blink
- Includes
- build
- deps
- images
- libs
- main.c
- LICENSE
- Makefile
- makefile.init
- README.md

PSoC 6 MCU: Empty Application Template

This is a minimal starter application template for PSoC® 6 MCU devices.

For instructions on modifying and using this application template to create applications, follow steps from the section "My First PSoC 6 MCU Design Using ModusToolbox IDE" in AN228571 – **Getting Started with PSoC 6 MCU on ModusToolbox.**

Requirements

- ModusToolbox™ software v2.1
- Programming Language: C
- Associated Parts: All PSoC® 6 MCU parts

Supported Toolchains (make variable

Outline: h1. PSoC 6 MCU: Empty; h2. Requirements; h2. Supported Toolc; h2. Supported Kits (r; h2. Related Resource; h2. Other Resources; h2. Document Histo

Eclipse IDE for ModusToolbox®

Start
- New Application
- Search Online for Code Examples
- Search Online for Libraries and BSPs
- Refresh Quick Panel

ch05a_ex01_blink (CY8CKIT-062S2-4
- Build ch05a_ex01_blink Application
- Clean ch05a_ex01_blink Application

Launches
- ch05a_ex01_blink Debug (JLink)
- ch05a_ex01_blink Debug (KitProg3_

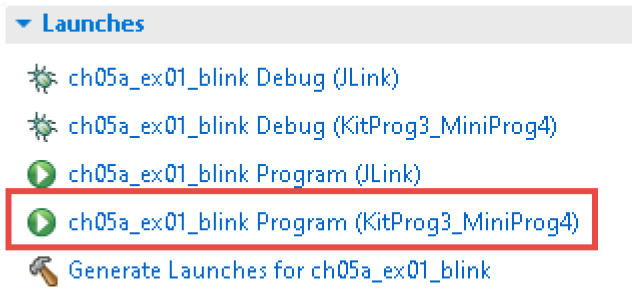
Console | Problems | Memory | News

ModusToolbox Console

☐ 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_MiniProg4)* from the Quick Panel Launches.



Launches

ch05a_ex01_blink Debug (JLink)

ch05a_ex01_blink Debug (KitProg3_MiniProg4)

ch05a_ex01_blink Program (JLink)

ch05a_ex01_blink Program (KitProg3_MiniProg4)

Generate Launches for ch05a_ex01_blink

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 "PSoC 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.

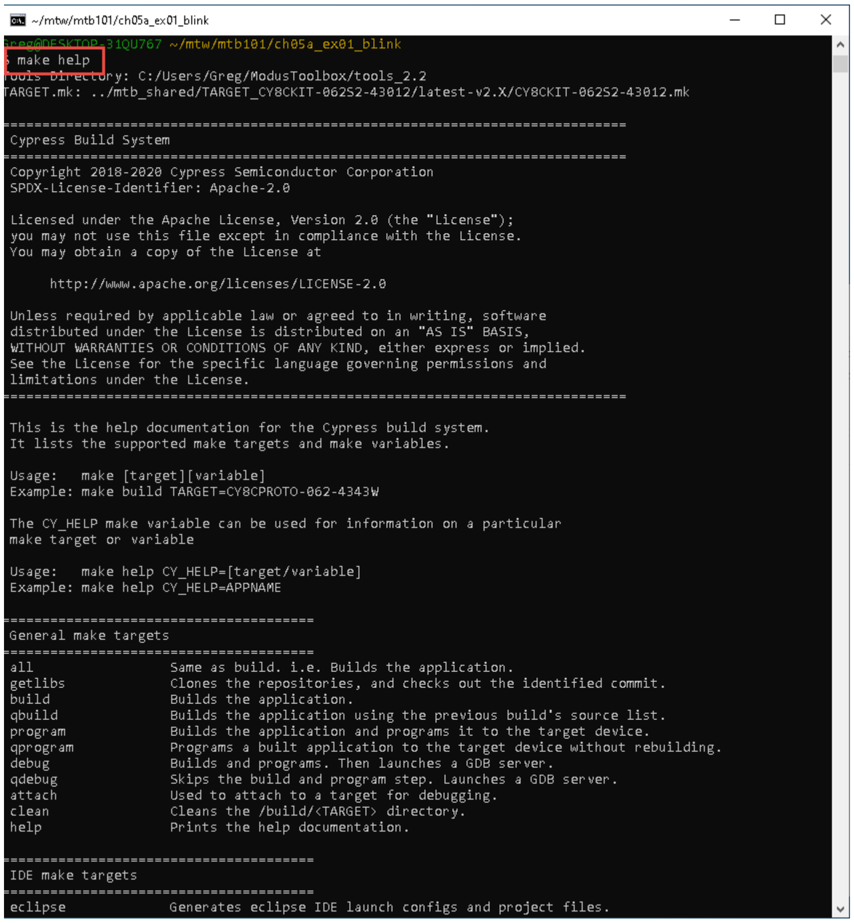
☐ 1. On Windows, select *modus-shell* from the list of programs (or search for it) in the Start menu.

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.



```
~/mtw/mtb101/ch05a_ex01_blink
Greg@DESKTOP-31QU767 ~/mtw/mtb101/ch05a_ex01_blink
$ make help
Tools Directory: C:/Users/Greg/ModusToolbox/tools_2.2
TARGET.mk: ../mtb_shared/TARGET_CY8CKIT-062S2-43012/latest-v2.X/CY8CKIT-062S2-43012.mk

==============================================================================
 Cypress Build System
==============================================================================
 Copyright 2018-2020 Cypress Semiconductor Corporation
 SPDX-License-Identifier: Apache-2.0

 Licensed under the Apache License, Version 2.0 (the "License");
 you may not use this file except in compliance with the License.
 You may obtain a copy of the License at

      http://www.apache.org/licenses/LICENSE-2.0

 Unless required by applicable law or agreed to in writing, software
 distributed under the License is distributed on an "AS IS" BASIS,
 WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 See the License for the specific language governing permissions and
 limitations under the License.
==============================================================================

 This is the help documentation for the Cypress build system.
 It lists the supported make targets and make variables.

 Usage:   make [target][variable]
 Example: make build TARGET=CY8CPROTO-062-4343W

 The CY_HELP make variable can be used for information on a particular
 make target or variable

 Usage:   make help CY_HELP=[target/variable]
 Example: make help CY_HELP=APPNAME

=======================================
 General make targets
=======================================
 all                     Same as build. i.e. Builds the application.
 getlibs                 Clones the repositories, and checks out the identified commit.
 build                   Builds the application.
 qbuild                  Builds the application using the previous build's source list.
 program                 Builds the application and programs it to the target device.
 qprogram                Programs a built application to the target device without rebuilding.
 debug                   Builds and programs. Then launches a GDB server.
 qdebug                  Skips the build and program step. Launches a GDB server.
 attach                  Used to attach to a target for debugging.
 clean                   Cleans the /build/<TARGET> directory.
 help                    Prints the help documentation.

=======================================
 IDE make targets
=======================================
 eclipse                 Generates eclipse IDE launch configs and project files.
```

☐ 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:

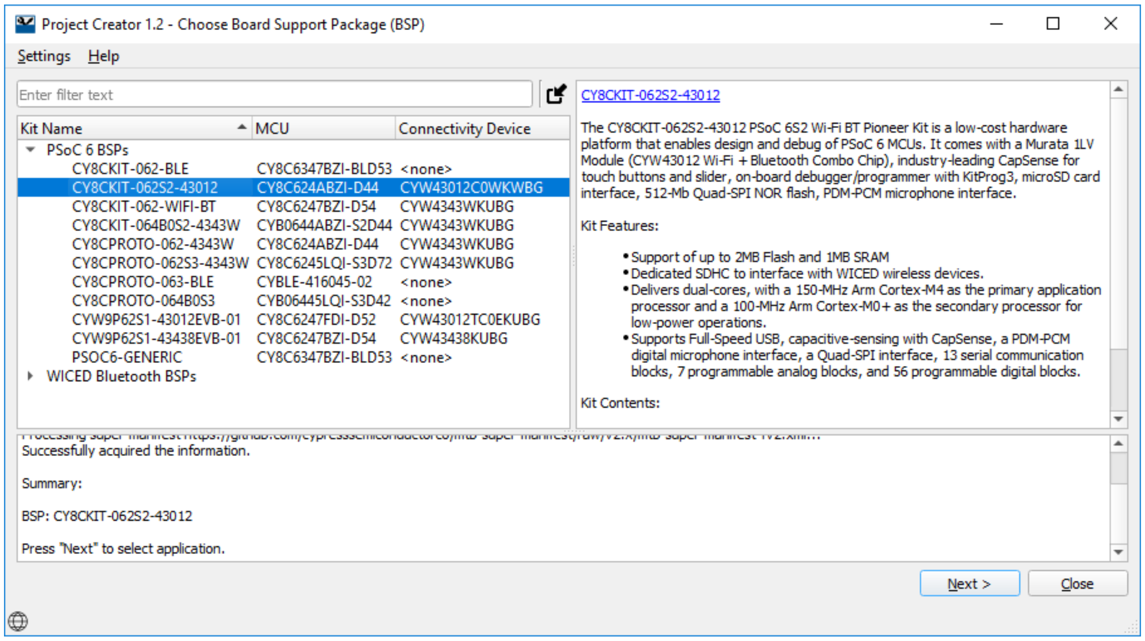
- Project Creator tool through Eclipse IDE (discussed previously)
- Stand-alone Project Creator tool
- Command Line / Manual Setup

You can also create projects in offline mode. See "Offline Content" section 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_2.2/project-creator* directory. You can also just search for "project-creator".

The first thing the project-creator tool does is read the configuration information from our GitHub site so that it knows all the BSPs etc. that we support. That information is stored in *manifest* files which we will discuss in the Manifests section.



Kit Name	MCU	Connectivity Device
PSoC 6 BSPs		
CY8CKIT-062-BLE	CY8C6347BZI-BLD53	<none>
CY8CKIT-062S2-43012	CY8C624ABZI-D44	CYW43012C0WKWBG
CY8CKIT-062-WIFI-BT	CY8C6247BZI-D54	CYW4343WKUBG
CY8CKIT-064B0S2-4343W	CYB0644ABZI-S2D44	CYW4343WKUBG
CY8CPROTO-062-4343W	CY8C624ABZI-D44	CYW4343WKUBG
CY8CPROTO-062S3-4343W	CY8C6245LQI-S3D72	CYW4343WKUBG
CY8CPROTO-063-BLE	CYBLE-416045-02	<none>
CY8CPROTO-064B0S3	CYB06445LQI-S3D42	<none>
CYW9P62S1-43012EVB-01	CY8C6247FDI-D52	CYW43012TC0EKUBG
CYW9P62S1-43438EVB-01	CY8C6247BZI-D54	CYW43438KUBG
PSOC6-GENERIC	CY8C6347BZI-BLD53	<none>
WICED Bluetooth BSPs		

CY8CKIT-062S2-43012

The CY8CKIT-062S2-43012 PSoC 6S2 Wi-Fi BT Pioneer Kit is a low-cost hardware platform that enables design and debug of PSoC 6 MCUs. It comes with a Murata 1LV Module (CYW43012 Wi-Fi + Bluetooth Combo Chip), industry-leading CapSense for touch buttons and slider, on-board debugger/programmer with KitProg3, microSD card interface, 512-Mb Quad-SPI NOR flash, PDM-PCM microphone interface.

Kit Features:

- Support of up to 2MB Flash and 1MB SRAM
- Dedicated SDHC to interface with WICED wireless devices.
- Delivers dual-cores, with a 150-MHz Arm Cortex-M4 as the primary application processor and a 100-MHz Arm Cortex-M0+ as the secondary processor for low-power operations.
- Supports Full-Speed USB, capacitive-sensing with CapSense, a PDM-PCM digital microphone interface, a Quad-SPI interface, 13 serial communication blocks, 7 programmable analog blocks, and 56 programmable digital blocks.

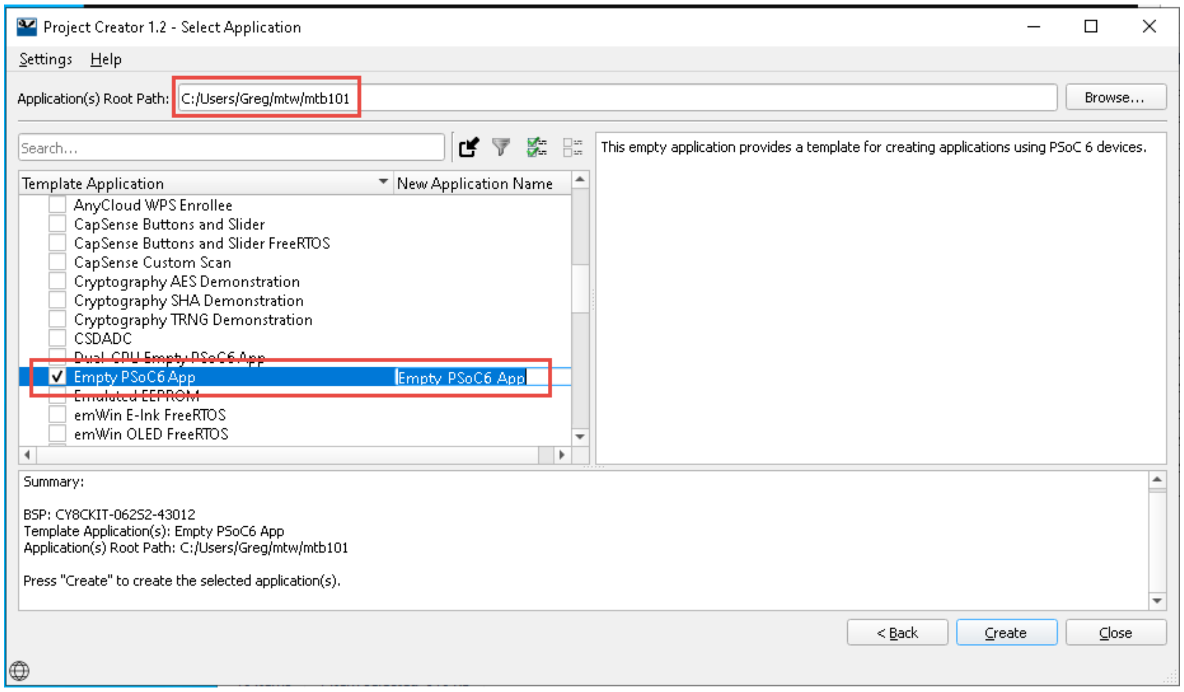
Kit Contents:

```
Processing super-manifest https://github.com/cypresssemiconductorco/mtb-super-manifest/raw/v2.x/mtb-super-manifest-fv2.xml...
Successfully acquired the information.

Summary:

BSP: CY8CKIT-062S2-43012

Press "Next" to select application.
```

Note that there is also an **Import** button so you can specify your own custom BSP if you have one. We will show you how to create your own BSP in the Creating your Own BSP section of this chapter.

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

Now you will be presented with all the code examples supported by the BSP you chose. Pick an application to start from and give it a name. You can specify a different **Application Root Path** if you don't want to use the default value. A directory with the name of your application(s) will be created inside the specified Application Root Path. The value for the Application Root Path is remembered from the previous invocation, so by default it will create applications in the same location that was used previously.


Project Creator 1.2 - Select Application

Application(s) Root Path: C:/Users/Greg/mtw/mtb101

This empty application provides a template for creating applications using PSoC 6 devices.

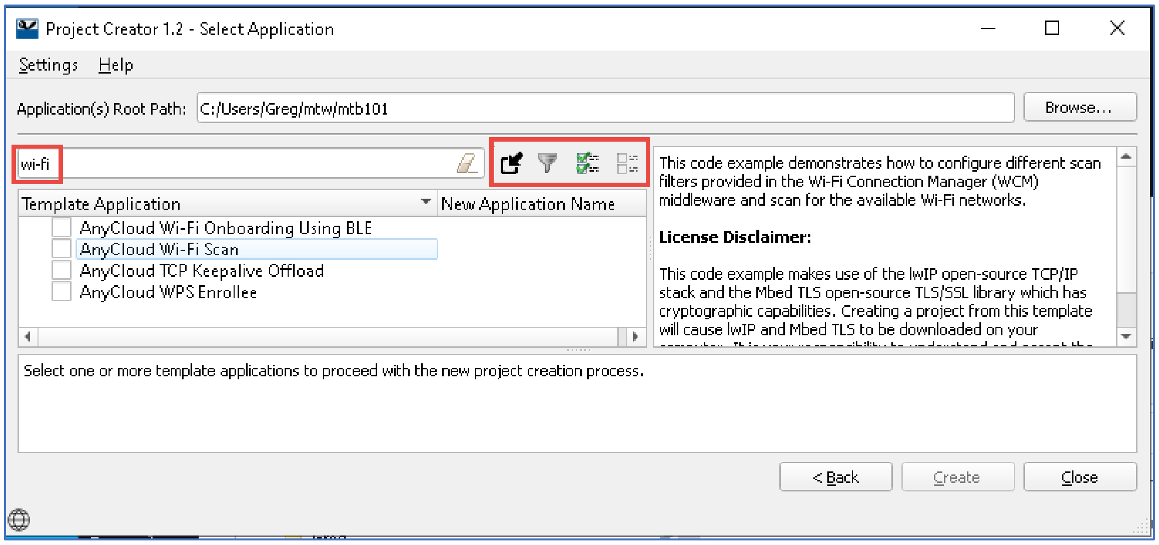
Template Application	New Application Name
AnyCloud WPS Enrollee	
CapSense Buttons and Slider	
CapSense Buttons and Slider FreeRTOS	
CapSense Custom Scan	
Cryptography AES Demonstration	
Cryptography SHA Demonstration	
Cryptography TRNG Demonstration	
CSDADC	
Dual-CPU Empty PSoC6 App	
✓ Empty PSoC6 App	Empty PSoC6 App
Emulated EEPROM	
emWin E-Ink FreeRTOS	
emWin OLED FreeRTOS	

Summary:

BSP: CY8CKIT-062S2-43012
Template Application(s): Empty PSoC6 App
Application(s) Root Path: C:/Users/Greg/mtw/mtb101

Press "Create" to create the selected application(s).

You can use the "Search…" box to enter a string to match from the application names and their descriptions. For example, if you enter wi-fi, you may see a list like this. The last 2 match because their descriptions have "wi-fi" in them. (The exact list will change over time as new code examples are created):


Project Creator 1.2 - Select Application

Application(s) Root Path: C:/Users/Greg/mtw/mtb101

wi-fi

Template Application	New Application Name
AnyCloud Wi-Fi Onboarding Using BLE	
AnyCloud Wi-Fi Scan	
AnyCloud TCP Keepalive Offload	
AnyCloud WPS Enrollee	

This code example demonstrates how to configure different scan filters provided in the Wi-Fi Connection Manager (WCM) middleware and scan for the available Wi-Fi networks.

License Disclaimer:

This code example makes use of the lwIP open-source TCP/IP stack and the Mbed TLS open-source TLS/SSL library which has cryptographic capabilities. Creating a project from this template will cause lwIP and Mbed TLS to be downloaded on your

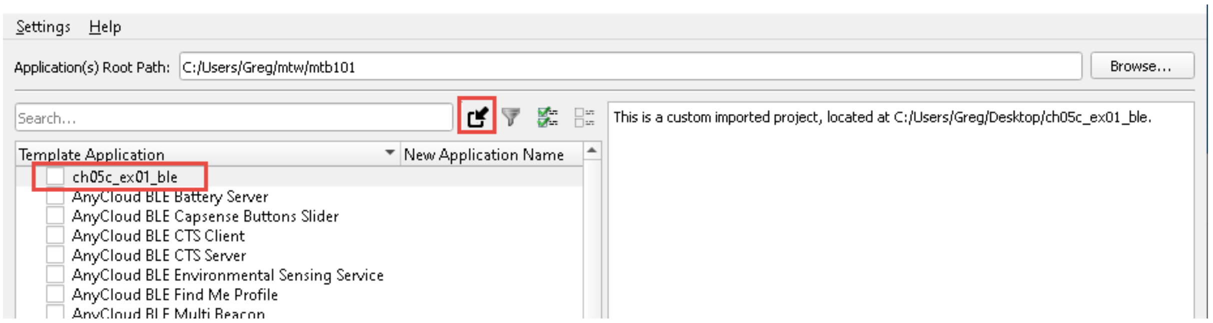
Select one or more template applications to proceed with the new project creation process.

If desired, you can select all the applications in the search result by using the **Select All** button or you can unselect them with the **Unselect All** button.

The **Filter** button can be used to list only the applications which are currently checked. This can be useful if you have a large number of applications checked and want to see them all listed together.

If you want to start with your own local project or template instead of one of the code examples that we provide, click the **Import** button. This allows you to browse files on your computer and then create a new application based on an existing 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.



Application(s) Root Path: C:/Users/Greg/mtw/mtb101

This is a custom imported project, located at C:/Users/Greg/Desktop/ch05c_ex01_ble.

ch05c_ex01_ble
AnyCloud BLE Battery Server
AnyCloud BLE Capsense Buttons Slider
AnyCloud BLE CTS Client
AnyCloud BLE CTS Server
AnyCloud BLE Environmental Sensing Service
AnyCloud BLE Find Me Profile
AnyCloud BLE Multi Beacon

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 mtb files which are a mechanism to include dependent libraries in an application.

Creating Bundled Applications

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 inside Eclipse. 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 into Eclipse, you will get a project called *myapps.myapp1* and *myapps.myapp2*.

PSoC 6 Dual-CPU applications use the bundled application concept. For those applications, there is a top-level application name with sub-application directories called *cm0p_app* and *cm4_app*. This will be discussed in more detail in Dual CPU Designs.

One thing to be careful of when creating bundled applications is the location of any shared libraries relative to the application. This will be discussed in more detail in the Library Management Flows section.

Once you have selected the application you want to create (whether it's one of our code examples or one of your own applications), 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**.

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_2.2/project-creator*) but the executable is called project-creator-cli.exe. You can run it with no arguments to see the different options available. For example, 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>
```

To use the Project Creator in CLI mode to create the empty PSoC 6 application for the kit we are using in the user's *mtw/mtb101* directory with a name of *my_app*, the command is:

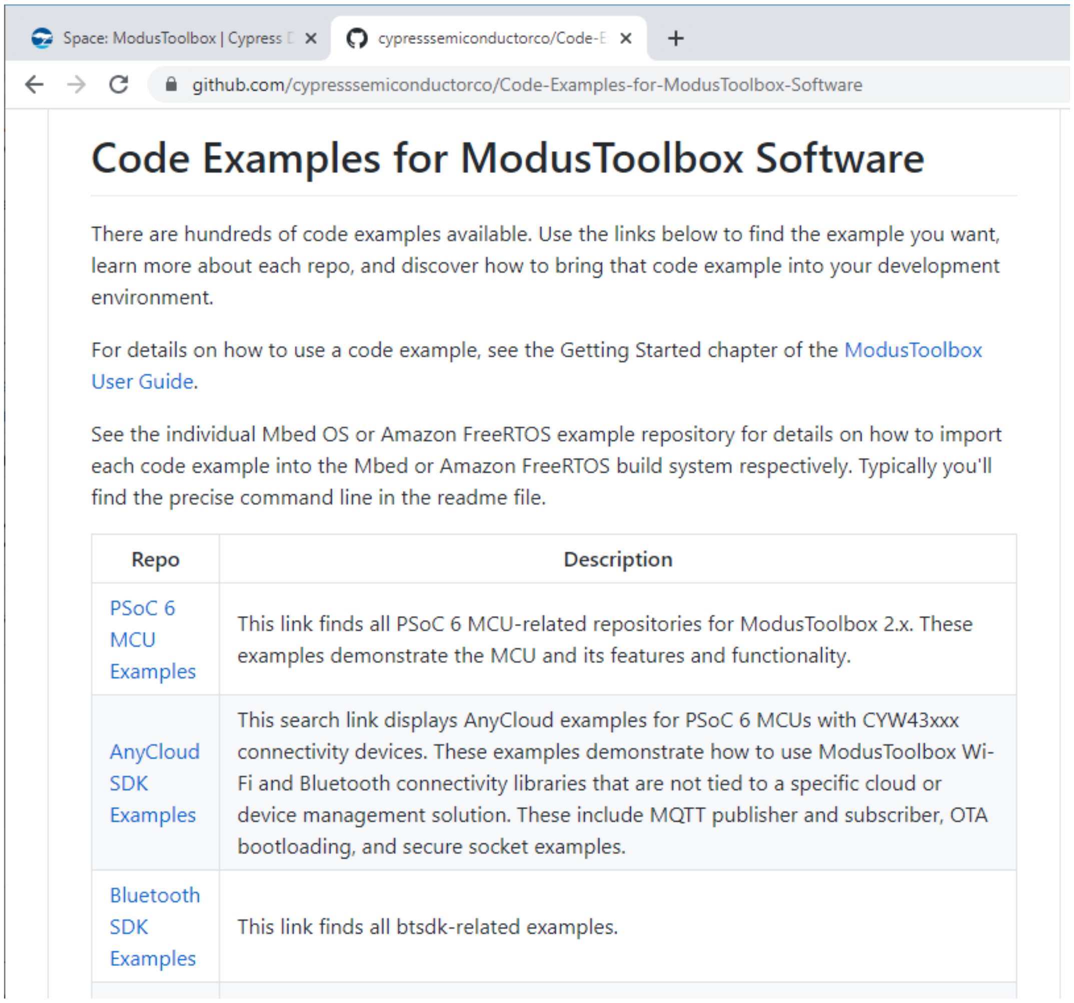
```
./project-creator-cli
      --board-id CY8CKIT-062S2-43012
      --app-id mtb-example-psoc6-empty-app
      --target-dir "~/mtw/mtb101"
      --user-app-name "my_app"
```

Note that double-quotes are required around the target directory name if you are using a tilde (~) to specify the user's home directory.

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.


Space: ModusToolbox | Cypress
cypresssemiconductorco/Code-E
github.com/cypresssemiconductorco/Code-Examples-for-ModusToolbox-Software

Code Examples for ModusToolbox Software

There are hundreds of code examples available. Use the links below to find the example you want, learn more about each repo, and discover how to bring that code example into your development environment.

For details on how to use a code example, see the Getting Started chapter of the ModusToolbox User Guide.

See the individual Mbed OS or Amazon FreeRTOS example repository for details on how to import each code example into the Mbed or Amazon FreeRTOS build system respectively. Typically you'll find the precise command line in the readme file.

Repo	Description
PSoC 6 MCU Examples	This link finds all PSoC 6 MCU-related repositories for ModusToolbox 2.x. These examples demonstrate the MCU and its features and functionality.
AnyCloud SDK Examples	This search link displays AnyCloud examples for PSoC 6 MCUs with CYW43xxx connectivity devices. These examples demonstrate how to use ModusToolbox Wi-Fi and Bluetooth connectivity libraries that are not tied to a specific cloud or device management solution. These include MQTT publisher and subscriber, OTA bootloading, and secure socket examples.
Bluetooth SDK Examples	This link finds all btsdk-related examples.

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:

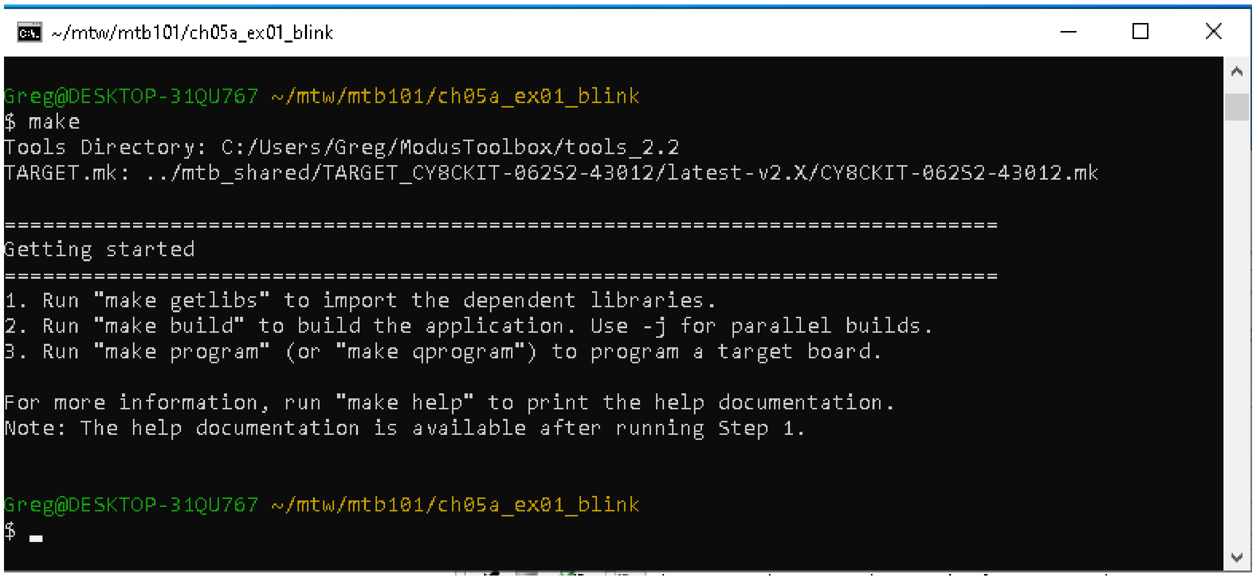
```
git clone https://github.com/cypresssemiconductorco/mtb-example-psoc6-
empty-app
```

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:



```
~/mtw/mtb101/ch05a_ex01_blink

Greg@DESKTOP-31QU767 ~/mtw/mtb101/ch05a_ex01_blink
$ make
Tools Directory: C:/Users/Greg/ModusToolbox/tools_2.2
TARGET.mk: ../mtb_shared/TARGET_CY8CKIT-062S2-43012/latest-v2.X/CY8CKIT-062S2-43012.mk

==============================================================================
Getting started
==============================================================================
1. Run "make getlibs" to import the dependent libraries.
2. Run "make build" to build the application. Use -j for parallel builds.
3. Run "make program" (or "make qprogram") to program a target board.

For more information, run "make help" to print the help documentation.
Note: The help documentation is available after running Step 1.


Greg@DESKTOP-31QU767 ~/mtw/mtb101/ch05a_ex01_blink
$
```

Then you can run:

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

Note If you use 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 Library Management for more details about adding and managing libraries.

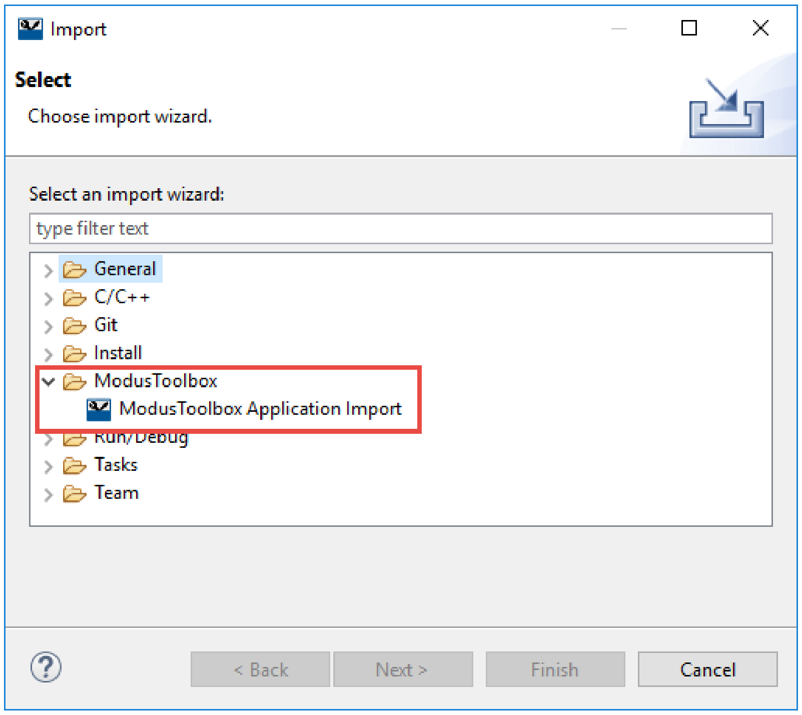
5a.5 Importing/Exporting Applications

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.

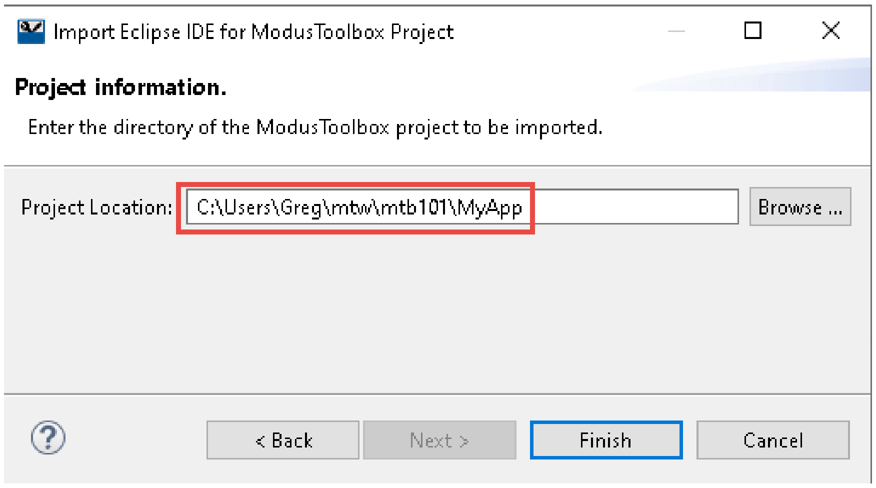
To open a command-line application in an IDE, you need to run a `make <IDE>` command to generate appropriate files. When you create an application starting from the Eclipse IDE, the `make eclipse` command is run for you automatically. There is also an import option in the Eclipse IDE for ModusToolbox 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… > ModusToolbox > 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**.


Import Eclipse IDE for ModusToolbox Project

Project information.

Enter the directory of the ModusToolbox project to be imported.

Project Location: C:\Users\Greg\mtw\mtb101\MyApp

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.

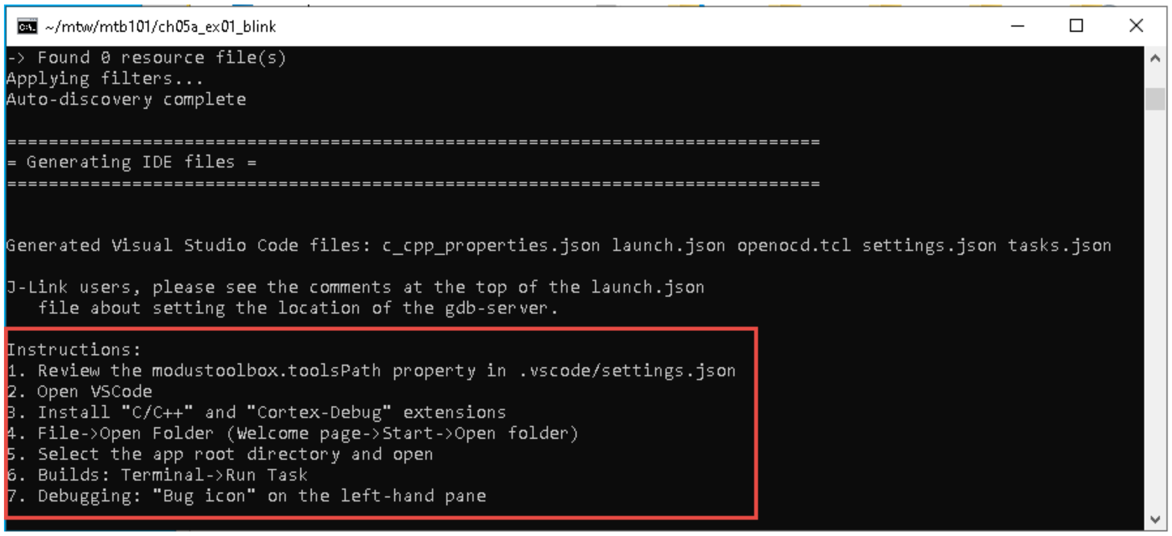
Some of the cases where you may end up with stale or multiple sets of launch configs:

- 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 complete 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:

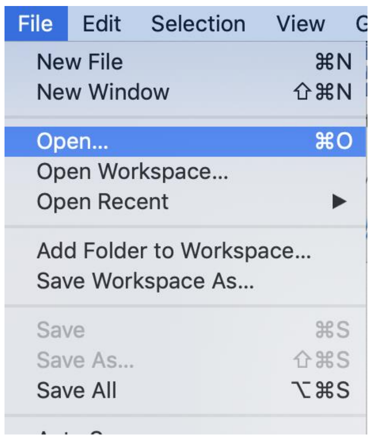
```
make vscode
```

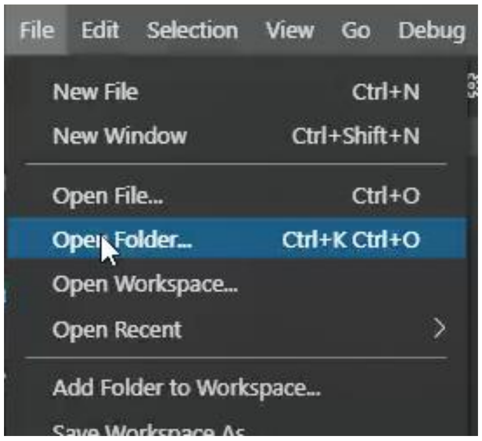


```
~/mtw/mtb101/ch05a_ex01_blink
-> Found 0 resource file(s)
Applying filters...
Auto-discovery complete

==============================================================================
= Generating IDE files =
==============================================================================


Generated Visual Studio Code files: c_cpp_properties.json launch.json openocd.tcl settings.json tasks.json

J-Link users, please see the comments at the top of the launch.json
   file about setting the location of the gdb-server.

Instructions:
1. Review the modustoolbox.toolsPath property in .vscode/settings.json
2. Open VSCode
3. Install "C/C++" and "Cortex-Debug" extensions
4. File->Open Folder (Welcome page->Start->Open folder)
5. Select the app root directory and open
6. Builds: Terminal->Run Task
7. Debugging: "Bug icon" on the left-hand pane
```

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

1. This step is just to verify that the version of ModusToolbox tools is what you expect.
2. Start VS Code.
3. Install the C/C++ and Cortex-Debug extensions if you don't already have them.
4. Open your application in Visual Studio Code using **File > Open** (on macOS) or **File > Open Folder** (on Windows).

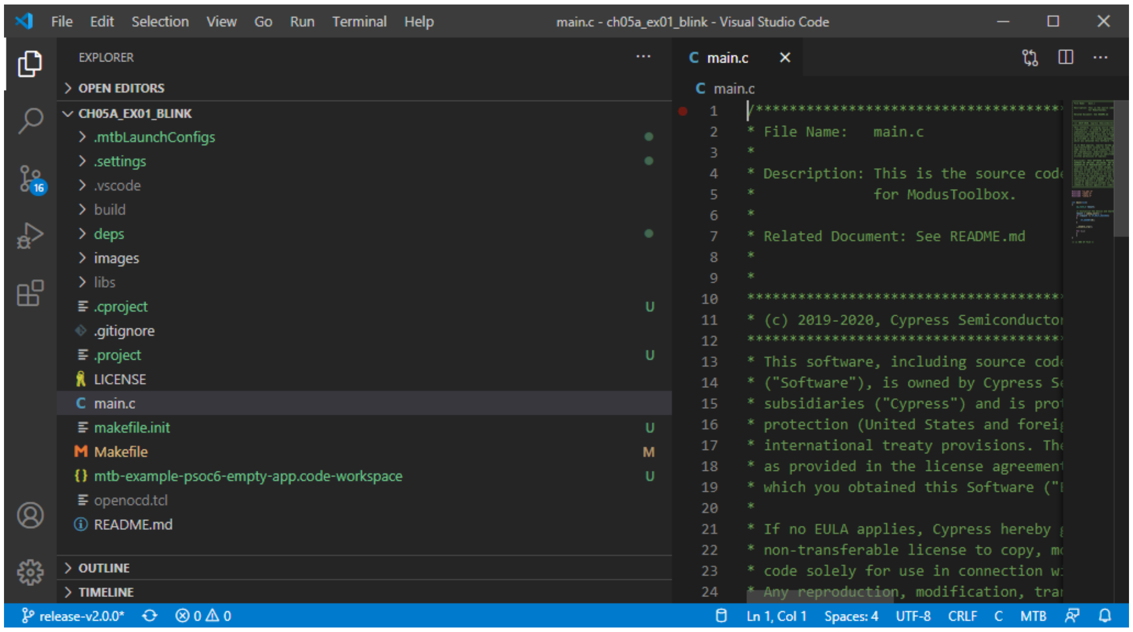




5. Then, navigate to the directory where your project resides and click **Open**.

   Hint: <u>Alternately, if it is in your path 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</u>. (Depending on how you installed VS Code, it may or may not be in your path by default.)

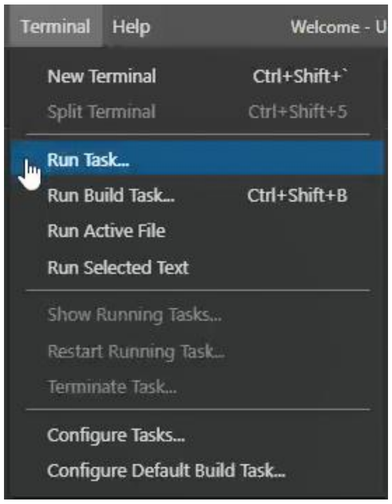
   Now your windows should look something like this. You can open the files by clicking on them in the Explorer window.



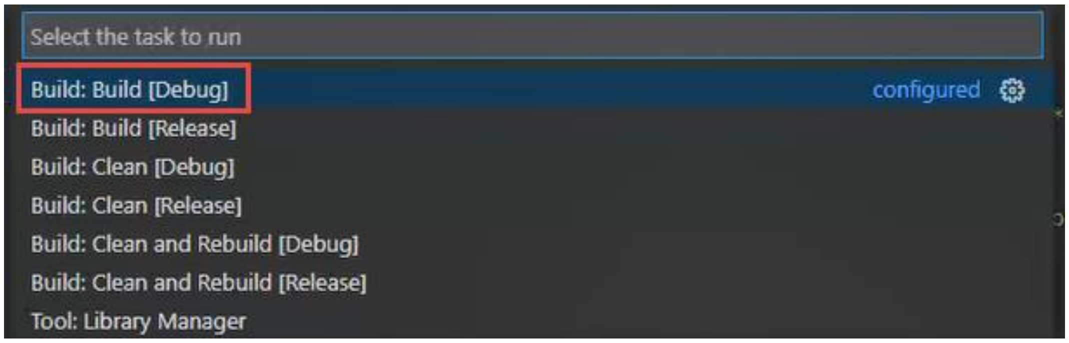
```
/**********************************
* File Name:   main.c
*
* Description: This is the source cod
*              for ModusToolbox.
*
* Related Document: See README.md
*
*
**********************************
* (c) 2019-2020, Cypress Semiconducto
**********************************
* This software, including source cod
* ("Software"), is owned by Cypress S
* subsidiaries ("Cypress") and is pro
* protection (United States and forei
* international treaty provisions. Th
* as provided in the license agreemen
* which you obtained this Software ("
*
* If no EULA applies, Cypress hereby
* non-transferable license to copy, m
* code solely for use in connection w
* Any reproduction, modification, tra
```

   <u>This method will work, but it does not show you the shared libraries (*mtb_shared*). If you want to see those libraries, follow the instructions in the next section to open a VS Code workspace</u>.

6. In order to build your project, make sure your application is selected and do **Terminal > Run Task…**

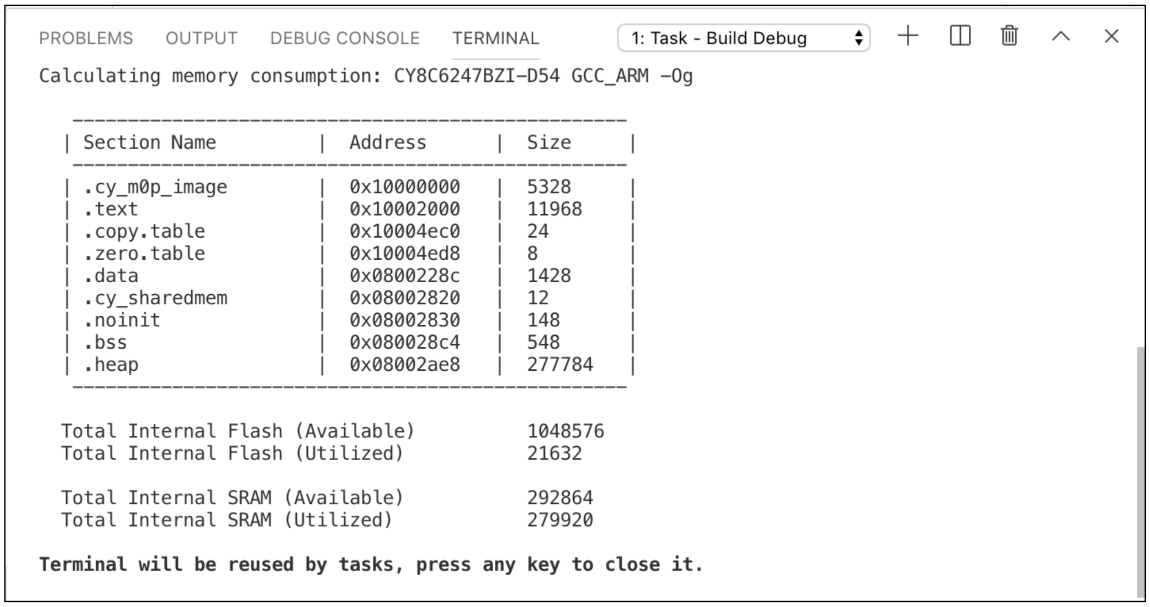


Then, select **Build Debug**. This will essentially run `make build` at the top level of your project.



(Note that you can also access the Library Manager from the Task list.)

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

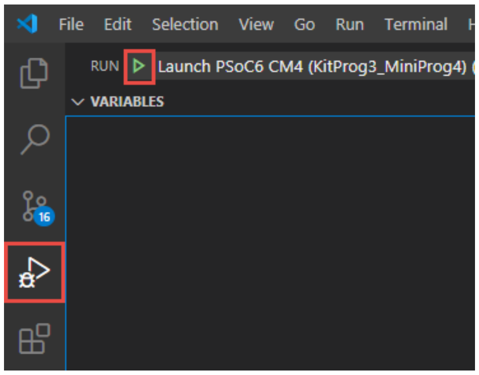


```
Calculating memory consumption: CY8C6247BZI-D54 GCC_ARM -Og

   --------------------------------------------------
  | Section Name          |  Address      |  Size    |
   --------------------------------------------------
  | .cy_m0p_image         |  0x10000000   |  5328    |
  | .text                 |  0x10002000   |  11968   |
  | .copy.table           |  0x10004ec0   |  24      |
  | .zero.table           |  0x10004ed8   |  8       |
  | .data                 |  0x0800228c   |  1428    |
  | .cy_sharedmem         |  0x08002820   |  12      |
  | .noinit               |  0x08002830   |  148     |
  | .bss                  |  0x080028c4   |  548     |
  | .heap                 |  0x08002ae8   |  277784  |
   --------------------------------------------------

  Total Internal Flash (Available)          1048576
  Total Internal Flash (Utilized)           21632

  Total Internal SRAM (Available)           292864
  Total Internal SRAM (Utilized)            279920

Terminal will be reused by tasks, press any key to close it.
```

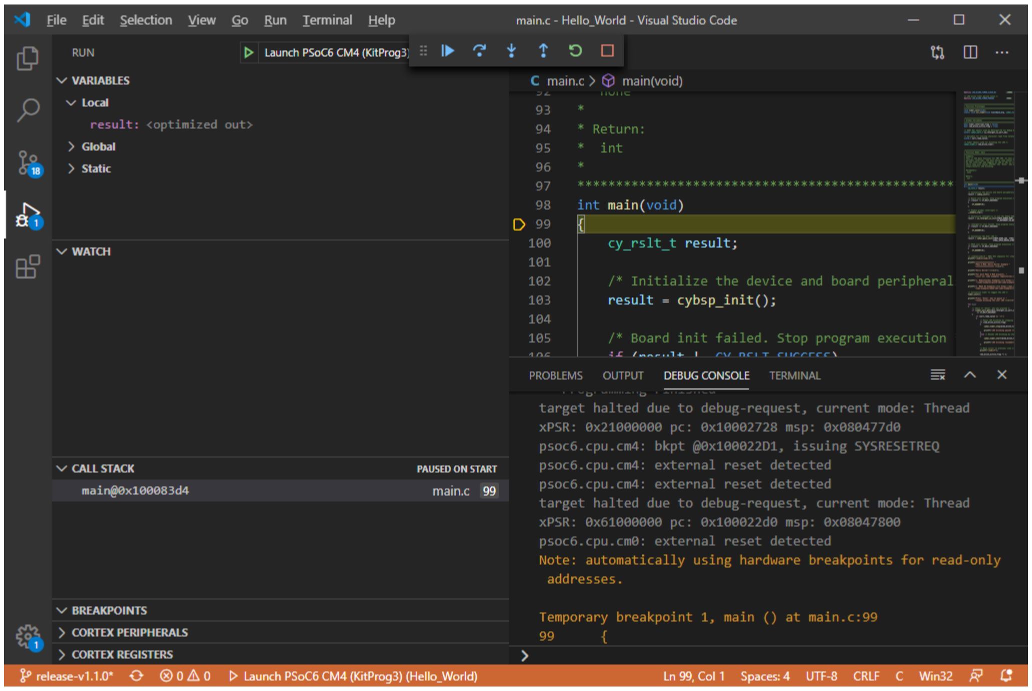
7. To program the kit or run the debugger, first make sure you have installed the “Cortex-Debug” VS Code extension from the marketplace. See the software installation instructions if you didn't do this when you installed VS Code.

   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.



```
result: <optimized out>
```

```
main@0x100083d4                main.c 99
```

```
 *
 * Return:
 *  int
 *
*************************************************
int main(void)
{
    cy_rslt_t result;

    /* Initialize the device and board peripheral
    result = cybsp_init();

    /* Board init failed. Stop program execution
```

```
target halted due to debug-request, current mode: Thread
xPSR: 0x21000000 pc: 0x10002728 msp: 0x080477d0
psoc6.cpu.cm4: bkpt @0x100022D1, issuing SYSRESETREQ
psoc6.cpu.cm4: external reset detected
psoc6.cpu.cm4: external reset detected
target halted due to debug-request, current mode: Thread
xPSR: 0x61000000 pc: 0x100022d0 msp: 0x08047800
psoc6.cpu.cm0: external reset detected
Note: automatically using hardware breakpoints for read-only
 addresses.

Temporary breakpoint 1, main () at main.c:99
99      {
```

Using Workspaces in VS Code

If you want to be able to see the shared libraries or even see all the applications that are in your application root path at once, you can use a VS Code workspace. They are text files with the extension code-workspace.

You can create and use workspaces in several different ways depending on what you want to see. You can launch the GUI first and then create a workspace or you can use the command line to launch the GUI and create the workspace as a single step.

If you want to see a single application along with the shared directory (e.g. application + mtb_shared) you can use a workspace that is created during `make vscode`. To use it:

> From the command line, go to the application's directory and run the following to launch the GUI and open the workspace:

```
code <workspace_name>
```

The workspace name ends in the extension *.code-workspace*.

> If you already have the GUI running, use the following to open the workspace:
> **File > Open Workspace…**
> Navigate to the workspace file and click **Open**

If you want to see all applications in your application root path:

> The easiest way to do this is to edit the workspace from one of your apps. The procedure is:

1. Copy/rename the workspace file from any application up one level (parallel with your apps and shared library repo).
2. Edit the workspace file to change the list of folders to match the list of applications and shared folder. You can add as many or as few applications as you want. Notice that the path to mtb_shared was changed. For example (only partial contents are shown):

   Original:
```
"folders": [
        {
                "path": "."
        },
        {
                "path": "../mtb_shared"
        }
],
```
   Modified:
```
"folders": [
        {
                "path": "ch05a_ex01_blink"
        },
        {
                "path": "ch05a_ex03_greem"
        },
        {
                "path": "mtb_shared"
        }
],
```

3. Once you have done this, open the workspace as usual (i.e. code <workspace name> from the command line or File > Open Workspace… from the GUI).

A few notes about workspaces:

1. You must run `make vscode` on any application that you include in your workspace so that it contains the required build information.
2. If more than one directory is open in a workspace, the Run Task list and the Debug Launch list will show entries for all available task/launches. Be careful to choose the correct entry.
3. If you quit and restart VS Code, any workspace that you didn't close will re-open. This may result in multiple instances of VS Code opening at the same time. To resolve this, use **File > Close Workspace**.
4. When you close a workspace that you have created, you will be asked if you want to save it. If you chose to save your workspace, you can open it using one of these methods:
    a. From the command line: `code <workspace_name>`
    b. From the GUI: **File > Open Workspace…**
5. If you create new applications after creating a workspace you can add them manually to your workspace file or if you have the workspace open in the GUI you can use **File > Add Folder to Workspace…)**.

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:

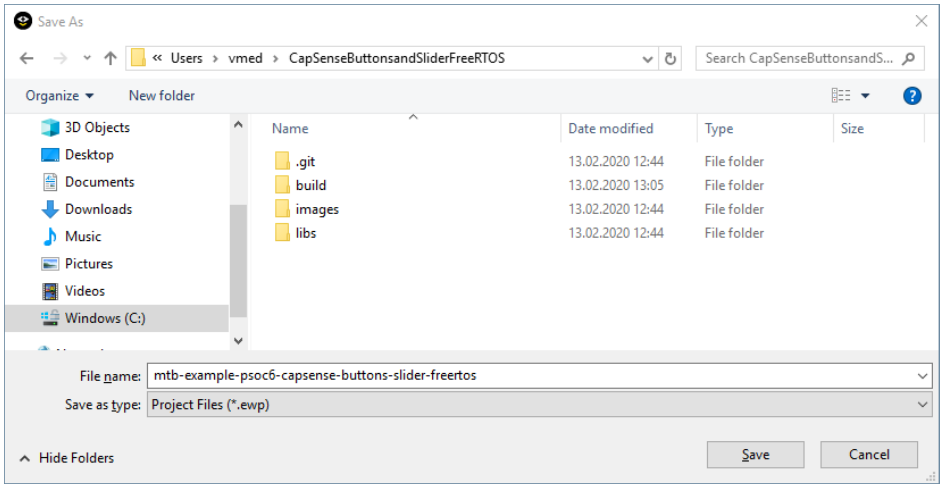
```
make ewarm8 TOOLCHAIN=IAR
```

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**.

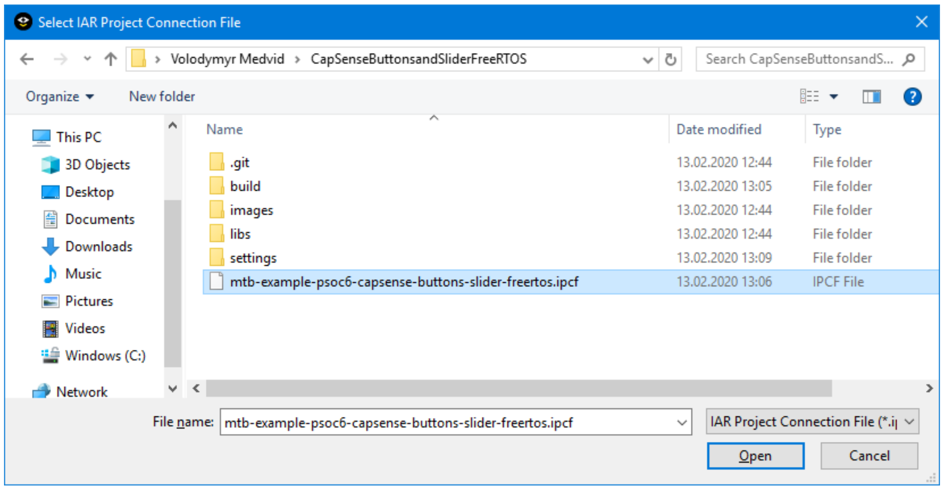


Name	Date modified	Type	Size
.git	13.02.2020 12:44	File folder	
build	13.02.2020 13:05	File folder	
images	13.02.2020 12:44	File folder	
libs	13.02.2020 12:44	File folder	

File name: mtb-example-psoc6-capsense-buttons-slider-freertos
Save as type: Project Files (*.ewp)

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**:


Select IAR Project Connection File
Volodymyr Medvid > CapSenseButtonsandSliderFreeRTOS
Search CapSenseButtonsandS...
Organize
New folder
This PC
3D Objects
Desktop
Documents
Downloads
Music
Pictures
Videos
Windows (C:)
Network
Open
Cancel

Name	Date modified	Type
.git	13.02.2020 12:44	File folder
build	13.02.2020 13:05	File folder
images	13.02.2020 12:44	File folder
libs	13.02.2020 12:44	File folder
settings	13.02.2020 13:09	File folder
mtb-example-psoc6-capsense-buttons-slider-freertos.ipcf	13.02.2020 13:06	IPCF File

File name: mtb-example-psoc6-capsense-buttons-slider-freertos.ipcf
IAR Project Connection File (*.ip

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 TOOLCHAIN=ARM
```

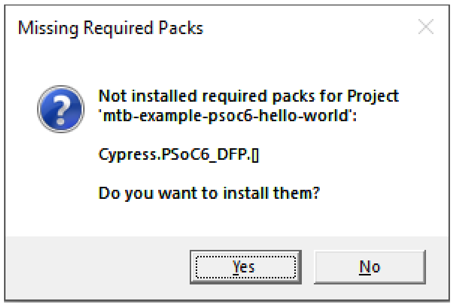
Note that export to Keil µVision is **not** currently supported for BT SoC solution applications.

This generates three files in the application directory:

- *<project-name>.cpdsc*
- *<project-name>.cprj*
- *<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:

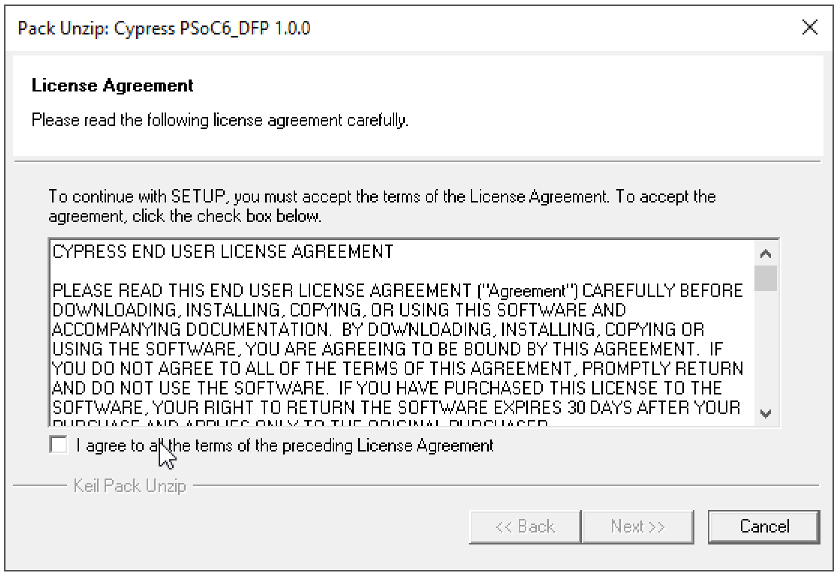

Missing Required Packs

Not installed required packs for Project 'mtb-example-psoc6-hello-world':

Cypress.PSoC6_DFP.[]

Do you want to install them?

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.


Pack Unzip: Cypress PSoC6_DFP 1.0.0

License Agreement

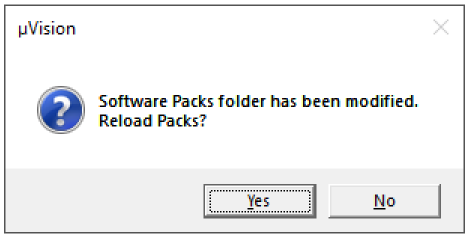
Please read the following license agreement carefully.

To continue with SETUP, you must accept the terms of the License Agreement. To accept the agreement, click the check box below.

CYPRESS END USER LICENSE AGREEMENT

PLEASE READ THIS END USER LICENSE AGREEMENT ("Agreement") CAREFULLY BEFORE DOWNLOADING, INSTALLING, COPYING, OR USING THIS SOFTWARE AND ACCOMPANYING DOCUMENTATION. BY DOWNLOADING, INSTALLING, COPYING OR USING THE SOFTWARE, YOU ARE AGREEING TO BE BOUND BY THIS AGREEMENT. IF YOU DO NOT AGREE TO ALL OF THE TERMS OF THIS AGREEMENT, PROMPTLY RETURN AND DO NOT USE THE SOFTWARE. IF YOU HAVE PURCHASED THIS LICENSE TO THE SOFTWARE, YOUR RIGHT TO RETURN THE SOFTWARE EXPIRES 30 DAYS AFTER YOUR

☐ I agree to all the terms of the preceding License Agreement


µVision

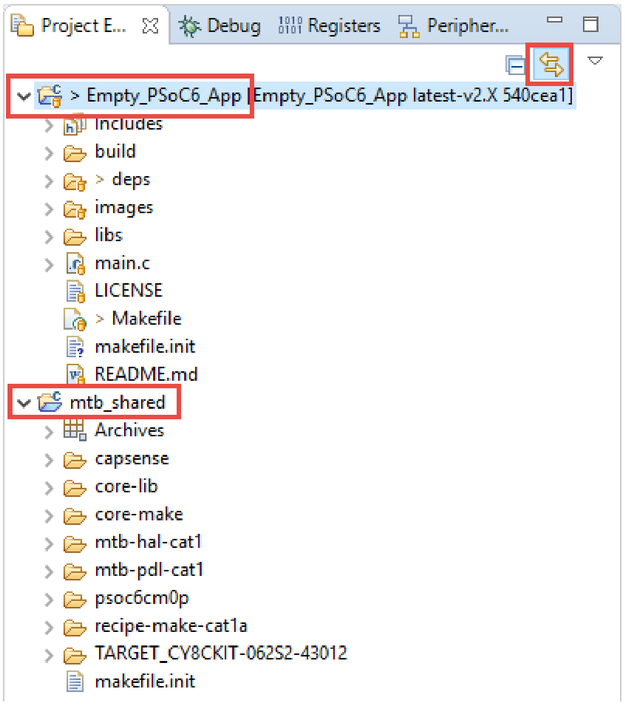
Software Packs folder has been modified.
Reload Packs?

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.



```
Project E...  Debug  Registers  Peripher...
> Empty_PSoC6_App [Empty_PSoC6_App latest-v2.X 540cea1]
    Includes
    build
    > deps
    images
    libs
    main.c
    LICENSE
    > Makefile
    makefile.init
    README.md
mtb_shared
    Archives
    capsense
    core-lib
    core-make
    mtb-hal-cat1
    mtb-pdl-cat1
    psoc6cm0p
    recipe-make-cat1a
    TARGET_CY8CKIT-062S2-43012
    makefile.init
```

Hint: if you are going back and forth between files from two different applications (e.g. to copy code from one application to another), switching between files can be greatly speeded up by turning off the "Link with Editor" feature. This prevents the active project from being changed every time you switch between files. It can be turned off using the icon with two arrows shown inside the red box above.

There are two top-level directories associated with most applications - the application project itself (Empty_PSoC6_App in this example) and a location that contains shared library source code (*mtb_shared*).

The directories in the application's project are:

5a.6.1 Binaries

Virtual directory that points to the elf file from the build. This folder will not appear until a build is done.

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. It will not appear until a build is done.

5a.6.4 deps

This directory contains *mtb* files that specify where the `make getlibs` command finds the libraries directly included by the project. Note that libraries included via *mtb* files may have their own library dependencies listed in the manifest files. We'll talk more about dependencies, *mtb* files and manifests in the Library Management section.

As you will see, the source code for libraries either go in the shared repository (for shared libraries) or in the *libs* directory inside the application (for libraries that are not shared).

5a.6.5 images

This directory contains artwork images used by the documentation.

5a.6.6 libs

The libs directory contains source code for local libraries (i.e. those that are not shared) and *mtb* files for indirect dependencies (i.e. libraries that are included by other libraries). Most libraries are placed in a shared repo and are therefore not in the *libs* directory. The *libs* directory also includes a file called *mtb.mk* which specified which shared libraries should be included in the application and where they can be found. This will be discussed in detail in the Library Management section.

As you will see, the *libs* directory only contains items that can be recreated by running `make getlibs`. Therefore, the libs directory should not be checked into a source control system.

5a.6.7 main.c

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

5a.6.8 LICENSE

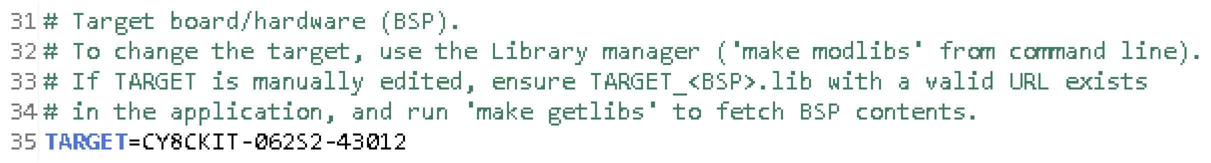
This is the ModusToolbox license agreement file.

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.

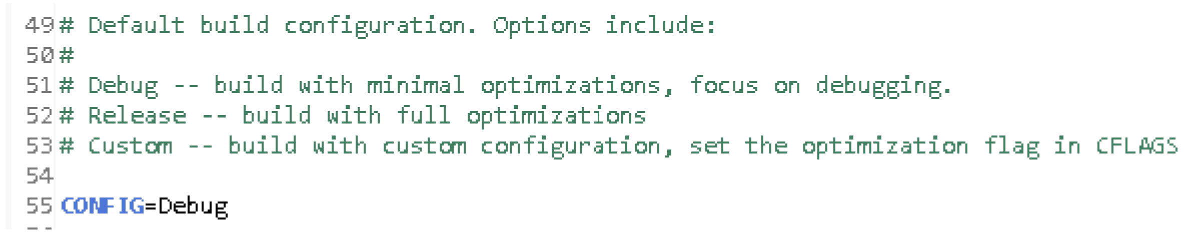
Various build settings can be set in the Makefile to change the build system behavior. These can be "make" settings or they can be settings that are passed to the compiler. Some examples are:

Target Device (`TARGET=`)



```
# Target board/hardware (BSP).
# To change the target, use the Library manager ('make modlibs' from command line).
# If TARGET is manually edited, ensure TARGET_<BSP>.lib with a valid URL exists
# in the application, and run 'make getlibs' to fetch BSP contents.
TARGET=CY8CKIT-062S2-43012
```

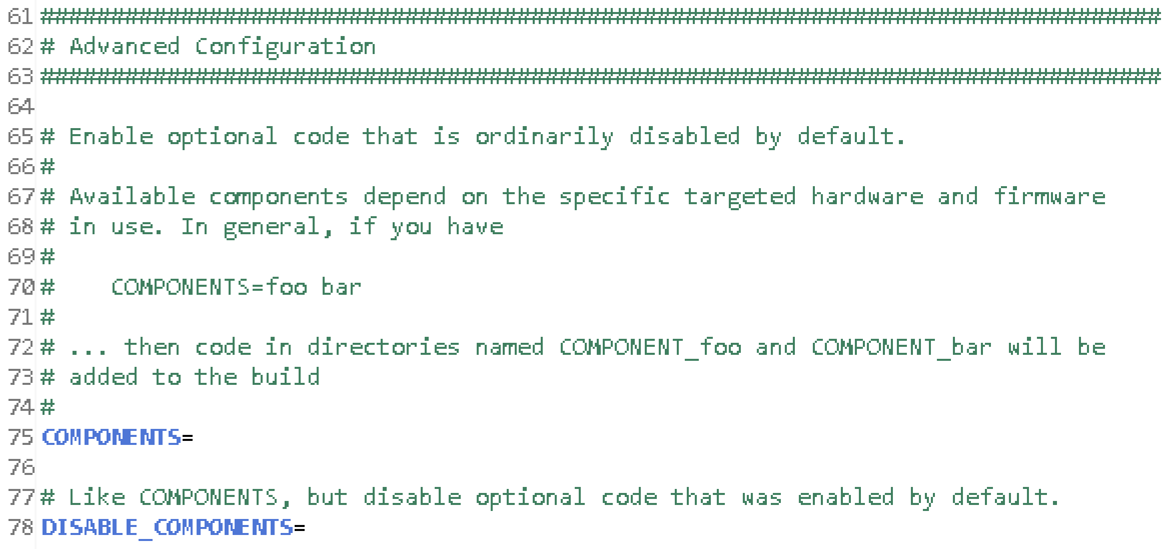
Build Configuration (CONFIG=)



```
# Default build configuration. Options include:
#
# Debug -- build with minimal optimizations, focus on debugging.
# Release -- build with full optimizations
# Custom -- build with custom configuration, set the optimization flag in CFLAGS

CONFIG=Debug
```

Note TARGET and CONFIG are used in the launch configurations (program, debug, etc.) so if you change either of these variables manually in the Makefile, you must update or regenerate the Launch configs inside the Eclipse IDE. Otherwise, you will not be programming/debugging the correct firmware.

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 Library Management

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.

5a.7.1 Library Classification

The way libraries are used in an application can be classified in several ways:

Shared vs. Local

Source code for libraries can either be stored locally in the application directory structure, or they can be placed in a location that can be shared between all the applications in a workspace.

Direct vs. Indirect

Libraries can either be referenced directly by your application (i.e. direct dependencies) or they can be pulled in as dependencies of other libraries (i.e. indirect dependencies). In many applications, the only direct dependency is a BSP. The BSP will include everything that it needs to work with the device such as the HAL and PDL. In other applications there will be additional direct dependencies such as Wi-Fi libraries or Bluetooth libraries.

Fixed vs. Dynamic Versions

You can specify an exact version of a library to use in your application, or you can specify a major release version with the intent that you will get the latest compatible version.

5a.7.2 Library Management Flows

Beginning with the ModusToolbox 2.2 release, we've developed a new way of structuring applications, called the MTB flow. Using this flow, applications can share Board Support Packages (BSPs) and libraries. If necessary, different applications can use different versions of the same shared library. Sharing resources reduces the number of files on your computer and speeds up subsequent application creation time. Shared BSPs, libraries and versions are stored in a new *mtb_shared* directory adjacent to your application directories.

You can easily switch a shared BSP or library to become local to a specific application, or back to being shared.

Looking ahead, most example applications will use the new MTB flow. However, there are still various applications that use the previous flow, now called the LIB flow, and these applications generally do not share BSPs and libraries. ModusToolbox fully supports both flows.

Note that the flow selection for a single application is a binary choice. If an application uses the MTB flow, only *mtb* files will be processed, and any *lib* files will be ignored.

MTB flow

First let's discuss the flow using *mtb* files. In this flow, the source code for all libraries will be put in a shared directory by default. (You'll see how to make them local using in the Library Manager section.) The default directory name is *mtb_shared* and its default location is parallel to the application directories (i.e. in the same application root path). The name and location of the shared directory can be customized using the Makefile variables `CY_GETLIBS_SHARED_NAME` and `CY_GETLIBS_SHARED_PATH`.

The `CY_GETLIBS_SHARED_PATH` variable will most commonly be changed for bundled apps. For most applications, the path is set to *../* to locate the directory one level up from the application directory (i.e. parallel to it). In the case of bundled applications, you will have one or more additional levels of hierarchy. If you have one extra level of hierarchy, the path would typically be set to *../../* so that the shared directory is in the usual place in the workspace.

Source code for local libraries will be placed in libs directory inside the application.

ModusToolbox knows about a library in your project in one of two ways depending on whether the library is a direct dependency or an indirect dependency that is included by another library.

For direct dependencies, there will be one or more *mtb* files somewhere in the directories of your project (typically in the deps directory but could be anywhere except the libs directory). An *mtb* file is simply a text file with the extension *.mtb* that has three fields separated by #:

- 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 path to where the library should be stored in the shared location (i.e. the directory path underneath *mtb_shared*).

A typical *mtb* file looks like this:

```
https://github.com/cypresssemiconductorco/TARGET_CY8CKIT-062S2-43012/#latest-
v1.X#$$ASSET_REPO$$/TARGET_CY8CKIT-062S2-43012/latest-v1.X
```

The variable `$$ASSET_REPO$$` points to the root of the shared location - it is specified in the application's Makefile. If you want a library to be local to the app instead of shared you can use `$$LOCAL$$` instead of `$$ASSET_REPO$$` in the *mtb* file before downloading the libraries. Typically, the version is excluded from the path for local libraries since there can only be one local version used in a given application. Using the above example, a library local to the app would normally be specified like this:

```
https://github.com/cypresssemiconductorco/TARGET_CY8CKIT-062S2-43012/#latest-
v1.X#$$LOCAL$$/TARGET_CY8CKIT-062S2-43012
```

Indirect dependencies for each library are found using information that is stored in a manifest file (more on that in the Manifests section). For each indirect dependency found, the Library Manager places an *mtb* file in the *libs* directory in the application.

Once all the *mtb* files are in the application, the `make getlibs` process (either called directly from the command line or by the Library Manager) finds the *mtb* files, pulls the libraries from the specified Git repos and stores them in the specified location (i.e. *mtb_shared/* for shared libraries and *libs/* for local libraries). Finally, a file called *mtb.mk* is created in the application's *libs* directory. That file is what the build system uses to find all the shared libraries required by the application.

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 *mtb* files by re-running `make getlibs`. This includes both shared libraries (in *mtb_shared*) and local libraries (in *libs*) - they all get pulled from Git when you run `make getlibs`.

The same is true for the *mtb* files for indirect references and the *mtb.mk* file which are also stored in the *libs* directory. In fact, the default *.gitignore* file in our code examples excludes the entire libs directory since you should not need to check in any files from that directory.

In summary, the default locations of files relative to the application root directory for the MTB flow are:

	direct / shared	**direct / local**	**indirect / shared**
.mtb file	./deps/	./deps/	./libs/
library source code	../mtb_shared/	./libs/	../mtb_shared/

LIB flow

Now, let's discuss the flow using *lib* files for completeness since there are still some applications that use this flow. In the LIB flow, all libraries are local to the app by default. ModusToolbox knows about a library in your project by having a lib file somewhere in the directories of your project (typically in the deps directory).

A *lib* file looks just like an *mtb* file except that the file extension is *.lib* and the path to the shared location is not specified. Instead it just has 2 fields:

- 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:

```
https://github.com/cypresssemiconductorco/TARGET_CY8CKIT-062S2-43012/#latest-v1.X
```

The `make getlibs` process (either called directly from the command line or by the Library Manager) pulls the libraries from the specified Git repo, but in this case, it searches for *lib* files instead of *mtb* files. The other difference is that all the library source code goes in the *libs* directory in the application itself instead of in a shared location. The *lib* files are found and processed recursively so that if a library that gets pulled in has dependencies, those dependencies are pulled in during the next pass.

Again, 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`.

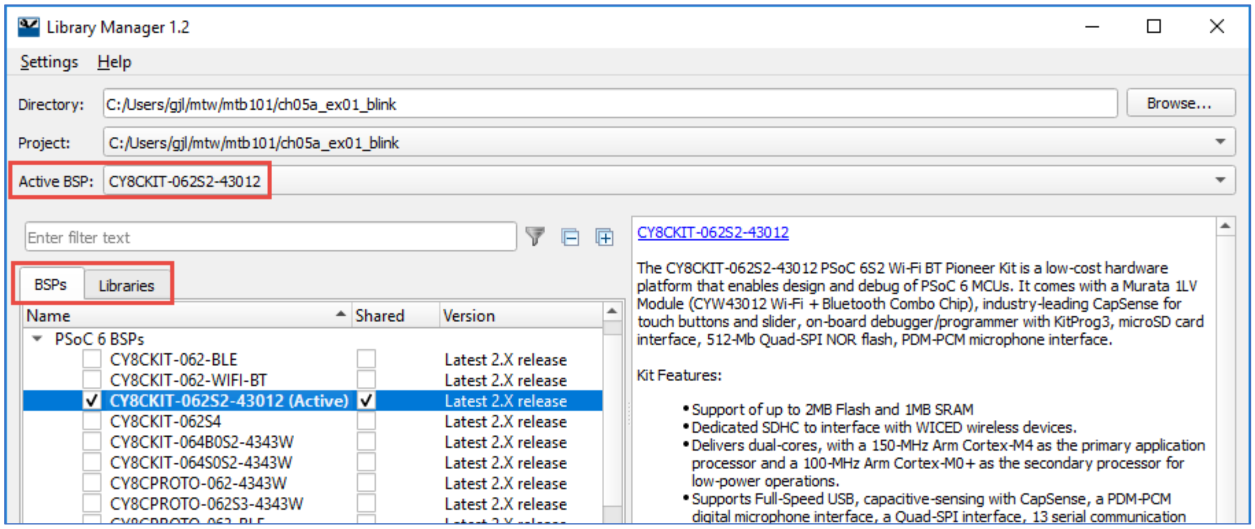
5a.7.3 Library Manager

ModusToolbox provides a GUI for helping you manage library files. You can run this from the Quick panel link “Library Manager”. It is available outside the IDE from ModusToolbox/tools_2.2/library-manager. If you are using the command line, you can launch it from an application directory using the command `make modlibs`.

The Library Manager knows where to find libraries and their dependencies by using manifest files, which will be discussed in the Manifests section. Therefore, *mtb* (or *lib* files for the LIB flow) 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 another source. You can create and include one or more custom manifest files for your custom libraries so that you can use the Library Manager, or you can manage them manually.

If, for some reason, you don't want to use the Library Manager GUI for a library that is in a manifest file, you can create an *mtb* file (or a *lib* file for the LIB flow) in the *deps* directory of your application and then run `make getlibs` from the root directory of your application. This will do the exact same thing that the Library Manager does.

The Library Manager GUI looks like this when using the MTB flow. The tabs and windows will be slightly different for the LIB flow, but we will focus on the MTB flow here.



Directory: C:/Users/gjl/mtw/mtb101/ch05a_ex01_blink
Project: C:/Users/gjl/mtw/mtb101/ch05a_ex01_blink
Active BSP: CY8CKIT-062S2-43012

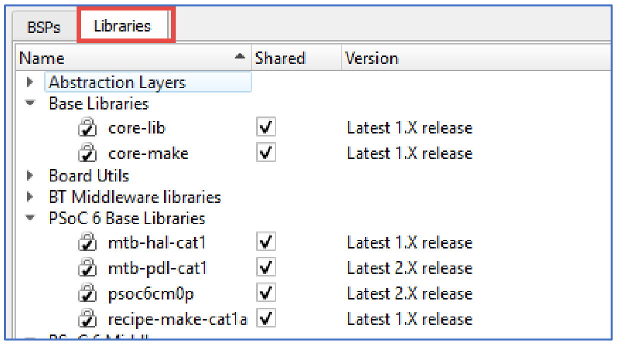
Name	Shared	Version
PSoC 6 BSPs		
☐ CY8CKIT-062-BLE	☐	Latest 2.X release
☐ CY8CKIT-062-WIFI-BT	☐	Latest 2.X release
✓ CY8CKIT-062S2-43012 (Active)	✓	Latest 2.X release
☐ CY8CKIT-062S4	☐	Latest 2.X release
☐ CY8CKIT-064B0S2-4343W	☐	Latest 2.X release
☐ CY8CKIT-064S0S2-4343W	☐	Latest 2.X release
☐ CY8CPROTO-062-4343W	☐	Latest 2.X release
☐ CY8CPROTO-062S3-4343W	☐	Latest 2.X release

CY8CKIT-062S2-43012

The CY8CKIT-062S2-43012 PSoC 6S2 Wi-Fi BT Pioneer Kit is a low-cost hardware platform that enables design and debug of PSoC 6 MCUs. It comes with a Murata 1LV Module (CYW43012 Wi-Fi + Bluetooth Combo Chip), industry-leading CapSense for touch buttons and slider, on-board debugger/programmer with KitProg3, microSD card interface, 512-Mb Quad-SPI NOR flash, PDM-PCM microphone interface.

Kit Features:

- Support of up to 2MB Flash and 1MB SRAM
- Dedicated SDHC to interface with WICED wireless devices.
- Delivers dual-cores, with a 150-MHz Arm Cortex-M4 as the primary application processor and a 100-MHz Arm Cortex-M0+ as the secondary processor for low-power operations.
- Supports Full-Speed USB, capacitive-sensing with CapSense, a PDM-PCM digital microphone interface, a Quad-SPI interface, 13 serial communication



Name	Shared	Version
Abstraction Layers		
Base Libraries		
core-lib	✓	Latest 1.X release
core-make	✓	Latest 1.X release
Board Utils		
BT Middleware libraries		
PSoC 6 Base Libraries		
mtb-hal-cat1	✓	Latest 1.X release
mtb-pdl-cat1	✓	Latest 2.X release
psoc6cm0p	✓	Latest 2.X release
recipe-make-cat1a	✓	Latest 1.X release

As you can see, the Library Manager has two tabs: *BSPs* (Board Support Packages) and *Libraries*.

The BSPs tab provides a mechanism to add/remove BSPs from the application and to specify which BSP (and version) should be used when building the application. An application can contain multiple BSPs, but a build from either the IDE or command line will build for and program to the “Active BSP” selected in the Library Manager. The Libraries tab allows you to add and remove libraries, as well as change their versions.

Shared Column: Both BSPs and Libraries can be "Shared" or not (i.e. "Local"). By default, most libraries are shared, meaning that the source code for the libraries is placed in the workspace's shared location (e.g. *mtb_shared*). If you uncheck the Shared box for a given library, its source code will be copied to the libs folder in the application itself. Note that this column only exists for the MTB flow.

Version Column: You can select a specific fixed version of a library for your application or choose a dynamic tag that points to a major version but allows `make getlibs` to fetch the latest minor version (e.g. Latest 1.X). The drop-down will list all available versions of the library.

Locked Items: A library shown with a "lock" symbol is an indirect dependency (meaning its *mtb* file is in the *libs* directory). An indirect dependency is included because another library requires it, so you cannot remove it from your application unless you first remove the library that requires it. You can change an indirect dependency from shared to local or you can change its version, but if you do it is converted to a direct dependency (meaning its *mtb* file is moved to the *deps* folder). This allows the local/version information to be retained by the application even if the *libs* directory is removed or is not checked into version control.

Behind the scenes, the Library Manager reads/creates/modifies *mtb* and *lib* files, creates/modifies the *mtb.mk* file, and then runs `make getlibs`. For example, when you change a library version and click *Update*, the Library Manager modifies the version specified in the corresponding *mtb* or *lib* file and then runs `make getlibs` to get the specified version. You can always do this for yourself from the command line.

Active BSP: When you change the Active BSP, the Library Manager edits the TARGET variable in the Makefile and then updates any Eclipse launch configs so that they point to the new BSP.

Eclipse IDE

Manually changing the TARGET in the Makefile will change project being built by the IDE, but it will **NOT** update the Eclipse launch configs, so it does not affect which hex file is copied to the kit. Therefore, it is recommended to use the Library Manager's Active BSP selection to change the target board if you are using the IDE to program/debug. If you do edit the Makefile manually, there is a link in the Quick Panel that says "Generate Launches for <app-name>" that you can use to fix them.

VSCode

In the case of VSCode, after updating the Active BSP or after manually editing the TARGET in the Makefile, you must update the appropriate launch configurations. You can re-run `make vscode` to regenerate the launch configurations, but keep in mind that this may over-write other VSCode settings, so it is safer to update them manually. When it is re-run, the `make vscode` process will create a backup copy of the previous files in *`.vscode/backup`*.

5a.7.4 Manual Library Management

If a library is not in a manifest file, it will not appear in the Library Manager and will need to be managed manually. Regardless of the library management flow, this is a two-step process:

1. Acquire the library
2. Acquire the library's dependencies

These steps can be done several different ways depending on the construction of the library and the library management flow being used by the application. Note that for (custom) BSPs, the Project Creator tool simplifies this process greatly. You can still use the other methods on BSPs if you want to include a custom BSP into an existing application. We will discuss how to create a custom BSP in Creating your Own BSP.

Note that by manually managing libraries, all dependencies will be included directly in the application. Indirect dependencies only occur in the MTB flow when libraries and dependencies are included in manifest files as described in the Library Management Flows section.

Acquiring a Library (MTB flow)

There are three basic ways to get a library that isn't in a manifest file using the MTB flow:

Requirements	**Method**
- Library is hosted in a Git repo	- Create an *mtb* file for the library in the application's deps directory. Run `make getlibs` from the application's root directory. - The library can be local or shared based on the *mtb* file contents.
- Any library - Library is local to app	- Use copy, git clone, or use some other revision control system to pull the library into the application. It can be located anywhere except the *libs* directory.
- Any library - Library is in a shared location	- Use copy, git clone, or use some other revision control system to pull the library into a shared location. - Edit the SEARCH variable in the application's Makefile to point to the library.
- Custom BSP	- Use copy, git clone, or use some other revision control system to get the BSP somewhere on disk if it isn't already. The location is not important. It will be copied into the application folder during project creation. - Use the Import function to select the BSP in Project Creator. The custom BSP will be copied into the application.

Once you have a library, the next step is to get its dependencies. The methods used depend on whether you are using the MTB flow or the LIB flow.

Acquiring Dependencies (MTB flow)

There are several ways to acquire dependencies:

Requirements	Method
- Library contains *mtbx* files for its dependencies - Dependencies can be local or shared	- Run `make import_deps` followed by `make getlibs` from the application's root directory. - (See `import_deps` details below)
- Library does not contain *mtbx* files but does contain lib files for its dependencies - Dependencies will be local	- Run `make lib2mtbx` , then `make import_deps` and finally `make getlibs` from the application's root directory. - (See `lib2mtbx` and `import_deps` details below)
- Dependencies are in a manifest	- Use the Library Manager.
- Dependencies are hosted in Git repos	- Create an *mtb* file for each dependency in the application's deps directory. - Run `make getlibs` from the application's root directory. - The dependencies can be local or shared based on the mtb file contents.
- Any Dependency - Dependency is local to app	- Use copy, git clone, or use some other revision control system to pull the library into the application. It can be located anywhere except the *libs* directory.
- Any dependency - Dependency is in a shared location	- Use copy, git clone, or use some other revision control system to pull the library into a shared location. - Edit the SEARCH variable in the application's Makefile to point to the library.
- Custom BSP	- Dependencies are pulled automatically by Project Creator.

The `make import_deps` target is intended specifically to acquire dependencies in MTB flow applications for libraries that don't have their dependencies specified manifest files. To use it, you need an *mtbx* file in the library for each dependency. The format of *mtbx* files is identical to *mtb* files. When you run `make import_deps`, it will copy the *mtbx* files from the library into the application's deps folder while also changing the extension to *mtb*. This results in the dependencies being direct to the application which are then pulled in when `make getlibs` is run. The libraries can either be local or shared depending on the contents of the *mtbx* files.

Usage:
```
make import_deps IMPORT_PATH=<path_to_library>
```

The `lib2mtbx` target is mainly intended for libraries (or BSPs) that were created for ModusToolbox 2.0 or 2.1 that do not contain *mtbx* files but do contain lib files for their dependencies. When you run `make lib2mtbx`, it will create *mtbx* files in the library based on the lib files in the library. The *mtbx* files will be created to place the dependencies local to the app unless the optional argument "shared" is specified. You can save the *mtbx* files to the library's source control so that you don't need to do this step the next time the library is included into an application.

Usage:
```
make lib2mtbx CONVERSION_PATH=<path_to_library> [CONVERSION_TYPE="shared"]
```

Acquiring a Library (LIB flow)

There are two basic ways to get a library that isn't in a manifest file using the LIB flow.

Requirements	Method
- Library is hosted in a Git repo	- Create a *lib* file for the library in the application's deps directory. - Run `make getlibs` from the application's root directory.
- Any library	- Use copy, git clone, or use some other revision control system to pull the library into the application. It can be located anywhere except the *libs* directory.
- Custom BSP	- Use copy, git clone, or use some other revision control system to get the BSP somewhere on disk if it isn't already. The location is not important. It will be copied into the application folder during project creation. - Use the Import function to select the BSP in Project Creator. The custom BSP will be copied into the application.

Acquiring Dependencies (LIB flow)

Again, there are several ways to acquire dependencies depending on the library's structure:

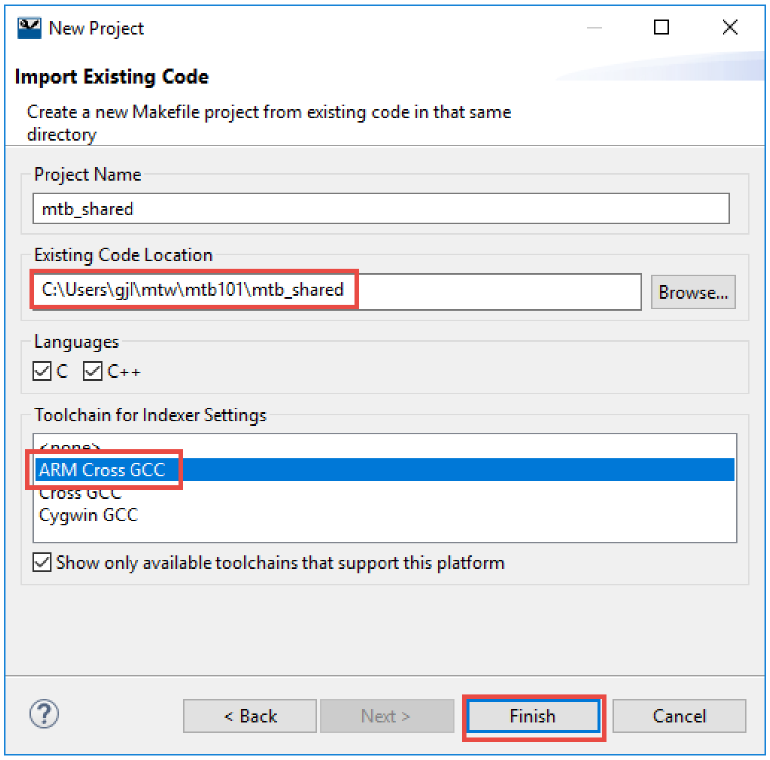
Requirements	Method
- Library contains lib files for its dependencies	- Run `make getlibs` (this step is already done if you used `make getlibs` to acquire the library).
- Dependencies are listed in a manifest	- Use the Library Manager.
- Dependencies are hosted in a Git repo	- Create a *lib* file for each dependency in the application's deps directory. - Run `make getlibs` from the application's root directory.
- Any dependency	- Use copy, git clone, or use some other revision control system to pull the library into the application. It can be located anywhere except the *libs* directory.
- Custom BSP	- Dependencies are pulled automatically by Project Creator.

5a.7.5 Re-Downloading Libraries

The local library directory (libs) and shared library repo (*mtb_shared*) can be deleted at any time since they can be re-created using either `make getlibs` from the command line or using **Update** in the Library Manager. Typically, those directories should NOT be checked into a revision control system since they should only contain libraries that are already controlled and versioned.

If you delete the *mtb_shared* directory from disk, you can easily re-acquire the libraries by using the Library Manager update function or by using `make getlibs` on an application, but the *mtb_shared* directory will not have Eclipse project information, so it may not be possible to open it from inside the Eclipse IDE and it may not work properly for Intellisense. To fix, follow these steps from inside Eclipse after regenerating *mtb_shared* using `make getlibs` or the Library Manager Update function.

1. From the Project Explorer window, right-click on *mtb_shared* and select **Delete**. Do NOT select the check box "Delete project contents on disk" (if you do, you will have to regenerate it again).
2. Select **File > Import > C/C++ > Existing Code as Makefile Project** and click **Next**.
3. **Browse** to the *mtb_shared* directory and click **Select Folder**.
4. The **Project Name** will be filled in automatically.
5. Select **ARM Cross GCC** for the toolchain and click **Finish**.


New Project

Import Existing Code

Create a new Makefile project from existing code in that same directory

Project Name: mtb_shared

Existing Code Location: C:\Users\gjl\mtw\mtb101\mtb_shared

Languages: ☑ C ☑ C++

Toolchain for Indexer Settings: <none>, [✓ ARM Cross GCC], Cross GCC, Cygwin GCC

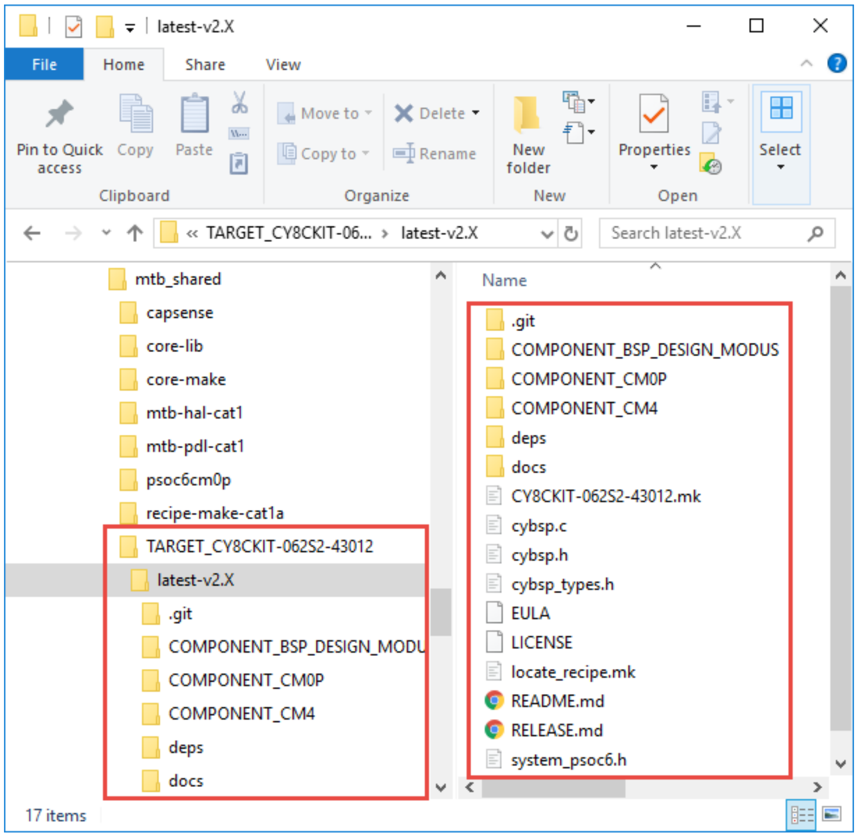
☑ Show only available toolchains that support this platform

6. To get IntelliSense to work again for an application you must re-build it first.

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 *mtb_shar*ed directory (by default if using the MTB flow):

5a.8.1 BSP Directory Structure

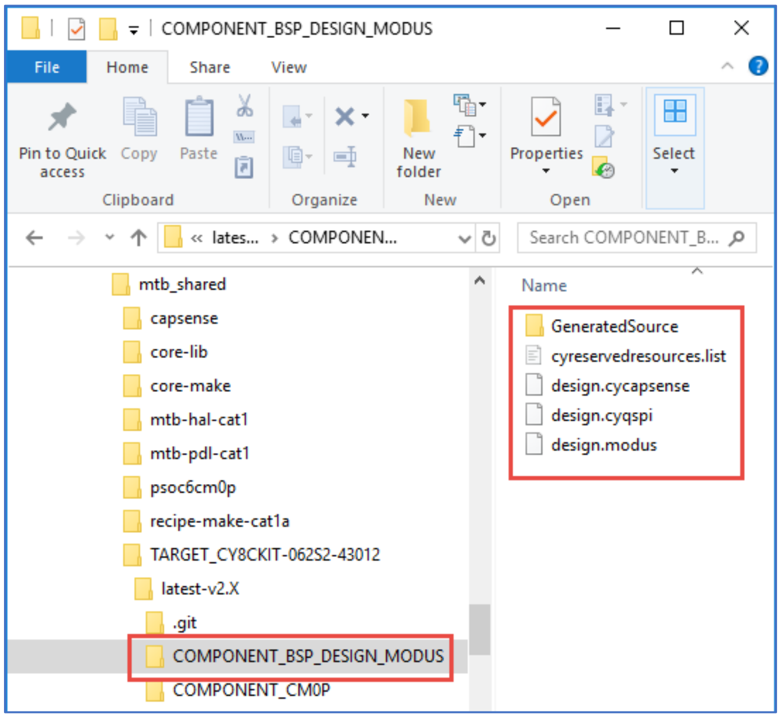


COMPONENT_BSP_DESIGN_MODUS

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.

Note that since these files are part of the BSP, if you use a configurator to modify them you will be modifying them for all applications that use the BSP (if it is shared). In addition, your changes will cause the BSP's repo to become dirty which means it cannot be updated to newer versions. Therefore, it is not recommended that you change any settings that are in the BSP. Instead, you can create a custom configuration for a single application (see Modifying the BSP Configuration (e.g. design.modus) for a Single Application) or you can create a custom BSP (see Creating your Own BSP).

A typical COMPONENT_BSP_DESIGN_MODUS directory looks like this:



mtb_shared
capsense
core-lib
core-make
mtb-hal-cat1
mtb-pdl-cat1
psoc6cm0p
recipe-make-cat1a
TARGET_CY8CKIT-062S2-43012
latest-v2.X
.git
COMPONENT_BSP_DESIGN_MODUS
COMPONENT_CM0P

GeneratedSource
cyreservedresources.list
design.cycapsense
design.cyqspi
design.modus

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. See BSP Documentation.

deps

The *deps* directory contains *lib* files for libraries that the BSP requires as dependencies. These are only used for the application flow that uses *lib* files. For the flow that uses *mtb* files, the dependency information is contained in a manifest file and the *lib* files are ignored.

cybsp_types.h

The *cybsp_types.h* file contains the aliases (macro definitions) for the board resources. It also contains comments for standard pin names so that they show up in the BSP documentation.

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.

locate_recipe.mk

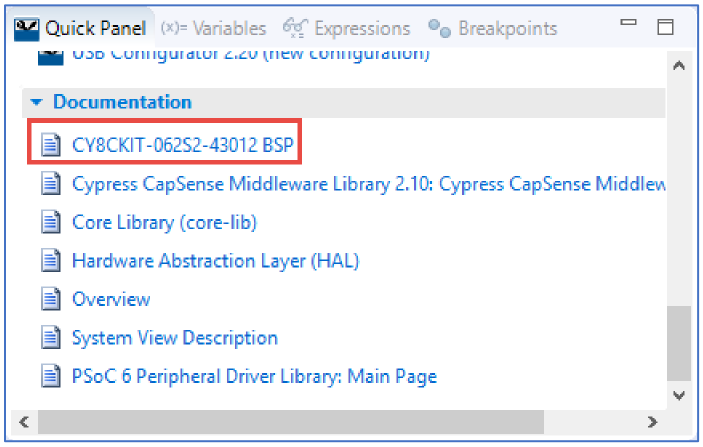
This file provides the path to the core and recipe make files that are needed by the build system.

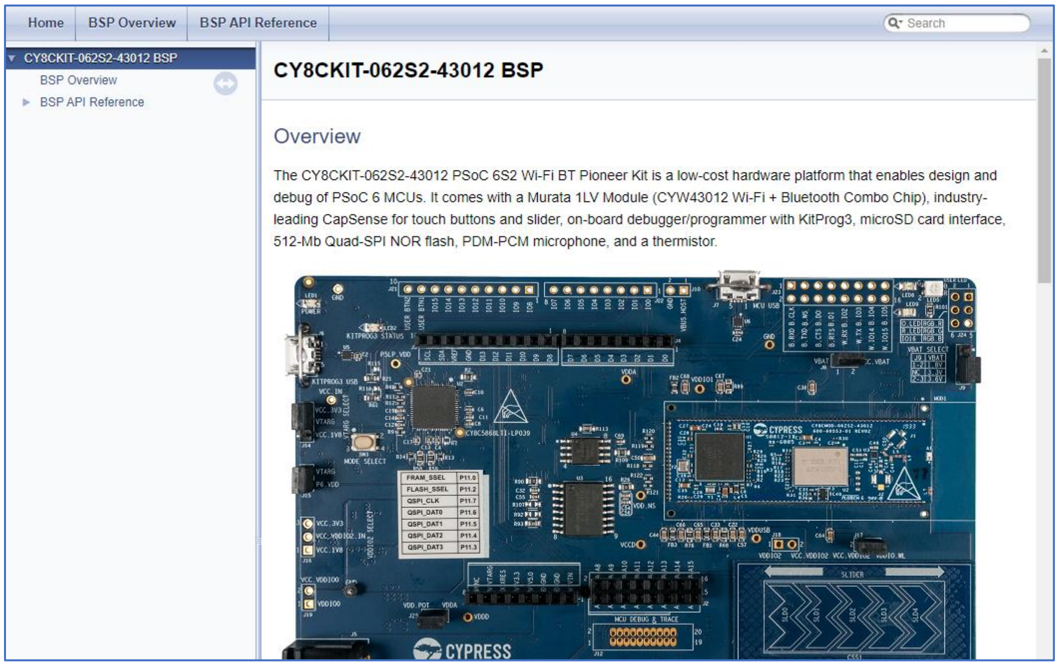
system_psoc6.h

This file contains device system header information.

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.


Quick Panel
(x)= Variables
Expressions
Breakpoints
USB Configurator 2.20 (new configuration)
Documentation
CY8CKIT-062S2-43012 BSP
Cypress CapSense Middleware Library 2.10: Cypress CapSense Middlew
Core Library (core-lib)
Hardware Abstraction Layer (HAL)
Overview
System View Description
PSoC 6 Peripheral Driver Library: Main Page



CY8CKIT-062S2-43012 BSP

Overview

The CY8CKIT-062S2-43012 PSoC 6S2 Wi-Fi BT Pioneer Kit is a low-cost hardware platform that enables design and debug of PSoC 6 MCUs. It comes with a Murata 1LV Module (CYW43012 Wi-Fi + Bluetooth Combo Chip), industry-leading CapSense for touch buttons and slider, on-board debugger/programmer with KitProg3, microSD card interface, 512-Mb Quad-SPI NOR flash, PDM-PCM microphone, and a thermistor.

5a.8.3 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), you should not modify the BSP directly since that results in changes to the BSP library. This will affect other applications in the same workspace if the BSP is shared, and it will prevent you from updating the BSP repository in the future. Instead, use the following process to create a custom set of configuration files for a specific application:

1. Create a directory at the root of the application to hold any custom BSP configuration files. For example:

   Hello_World/COMPONENT_CUSTOM_DESIGN_MODUS

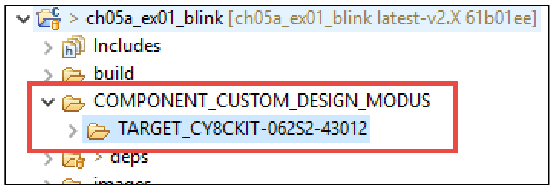
   This is a recommended best practice. In an upcoming step, you will modify the Makefile to include files from that directory instead of the directory containing the default configuration files in the BSP.

2. Create a subdirectory for each target that you want to support in your application. For example:

   Hello_World/COMPONENT_CUSTOM_DESIGN_MODUS/TARGET_CY8CKIT-062S2-43012

   The subdirectory name must be *TARGET_<board name>*. Again, this is a recommended best practice. If you only ever build with one BSP, this directory is not required, but it is safer to include it.

   The build system automatically includes all source files inside a directory that begins with *TARGET_*, followed by the target name for compilation, when that target is specified in the application's makefile. The file structure appears as follows. In this example, the *COMPONENT_BSP_DESIGN_MODUS* directory for this application is overridden for just one target: CY8CKIT-062S2-43012.



```
ch05a_ex01_blink [ch05a_ex01_blink latest-v2.X 61b01ee]
  Includes
  build
  COMPONENT_CUSTOM_DESIGN_MODUS
    TARGET_CY8CKIT-062S2-43012
  deps
```

3. Copy the *design.modus* file and other configuration files (that is, everything from inside the original BSP's *COMPONENT_BSP_DESIGN_MODUS* directory), and paste them into the new directory for the target.
4. In the application's Makefile, add the following lines. For example:

   ```
   DISABLE_COMPONENTS += BSP_DESIGN_MODUS

   COMPONENTS += CUSTOM_DESIGN_MODUS
   ```

   Note The Makefile already contains blank `DISABLE_COMPONENTS and COMPONENTS` lines where you can add the appropriate names.

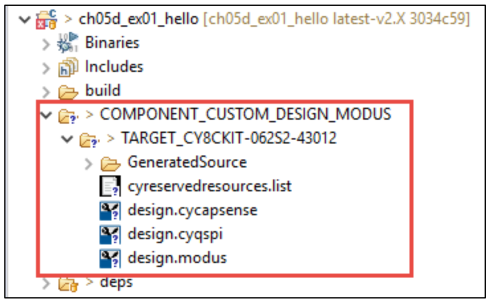
   The first line disables the configuration files from the original BSP since they are now in different directory.

The second line is required to specify the new directory and include your custom configuration files 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 set up in the configuration files.

5. Customize the configuration files as required, such as using the Device Configurator to open the *design.modus* file and modify appropriate settings.

   Note When you first create a custom configuration for an application, the Eclipse IDE Quick Panel entry to launch the Device Configurator may still open the *design.modus* file from the original BSP instead of the custom file. To fix this, click the **Refresh Quick Panel** link.

   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:



```
> ch05d_ex01_hello [ch05d_ex01_hello latest-v2.X 3034c59]
  > Binaries
  > Includes
  > build
  > COMPONENT_CUSTOM_DESIGN_MODUS
    > TARGET_CY8CKIT-062S2-43012
      > GeneratedSource
        cyreservedresources.list
        design.cycapsense
        design.cyqspi
        design.modus
  > deps
```

6. When finished customizing the configuration settings, you can build the application and program the device as usual.

5a.8.4 Creating your Own BSP

If you want to change more than just the configuration from the *COMPONENT_BSP_DESIGN_MODUS* directory (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 custom BSP, do the following:

1. Locate the closest-matching BSP to your intended custom BSP and set that as the default `TARGET` for the application in the Makefile.
2. In the application directory, run the `make bsp` target.

   Specify the new board name by passing the value to the `TARGET_GEN` variable. Optionally you may specify a new device (`DEVICE_GEN`) and additional devices (`ADDITIONAL_DEVICES_GEN`) if they are different from the BSP that you started from. For example:

```
make bsp TARGET_GEN=MyBSP DEVICE_GEN=CY8C624ABZI-S2D44
ADDITIONAL_DEVICES_GEN=CYW4343WKUBG
```

   This command creates a new BSP with the provided name at the top of the application project. It automatically copies the relevant startup and linker scripts into the newly created BSP, based on the device specified by the `DEVICE_GEN` option.

It also creates *.mtbx* files for all the BSP's dependences. The Project Creator tool uses these files when you import your custom BSP into that tool. These files can also be used with the `make import_deps` command if you manually include the custom BSP in a new application (see Manual Library Management).

Note 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. You can delete these from the BSP. Be sure to remove the corresponding *.mtbx* files as well.

Note If you want your custom BSP to support only the LIB flow, then you should manually remove the *.mtbx* files from the BSP after creating it.

3. Open the Device Configurator to customize settings in the new device's *design.modus* file for pin names, clocks, power supplies, and peripherals as required. Address any issues that arise.
4. Update the application's Makefile `TARGET` variable to point to your new BSP.
5. If using an IDE, regenerate the configuration settings to reflect the new BSP. Use the appropriate command(s) for the IDE(s) that are being used. For example: `make vscode`.
6. When you want to use a custom BSP in a new application, the easiest method to include it is to use the Import functionality in Project Creator. You can also use other manual library management techniques if you prefer - see Manual Library Management.

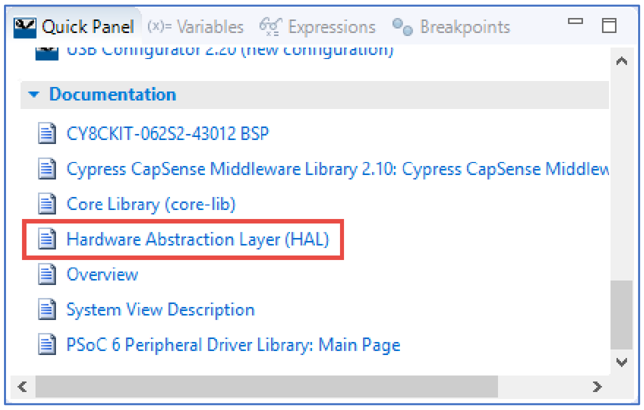
Note Use make help to see all supported IDE make targets. See also Importing/Exporting Applications in this document.

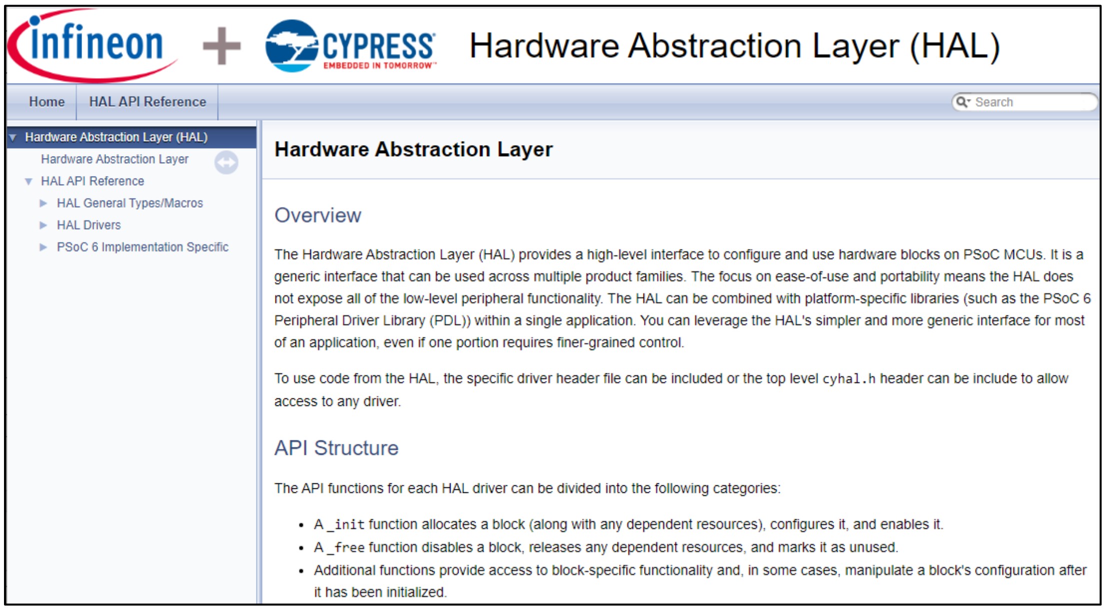
If you want to re-use a custom BSP on multiple applications, you can copy it into each application or you can put it into a version control system such as Git. See Manifests for information on how to create a manifest to include your custom BSPs and their dependencies if you want them to show up as standard BSPs in the Project Creator and Library Manager.

5a.9 HAL

The Hardware Abstraction Layer (HAL) provides a high-level interface to configure and use hardware blocks on our 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 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."


Quick Panel
(x)= Variables
Expressions
Breakpoints
USB Configurator 2.20 (new configuration)
Documentation
CY8CKIT-062S2-43012 BSP
Cypress CapSense Middleware Library 2.10: Cypress CapSense Middlew
Core Library (core-lib)
Hardware Abstraction Layer (HAL)
Overview
System View Description
PSoC 6 Peripheral Driver Library: Main Page


Hardware Abstraction Layer (HAL)
Home
HAL API Reference
Search
Hardware Abstraction Layer (HAL)
Hardware Abstraction Layer
HAL API Reference
HAL General Types/Macros
HAL Drivers
PSoC 6 Implementation Specific

Hardware Abstraction Layer

Overview

The Hardware Abstraction Layer (HAL) provides a high-level interface to configure and use hardware blocks on PSoC 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.

To use code from the HAL, the specific driver header file can be included or the top level `cyhal.h` header can be include to allow access to any driver.

API Structure

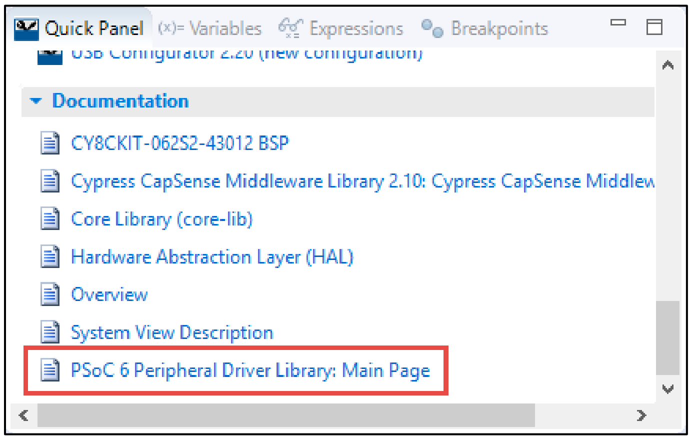
The API functions for each HAL driver can be divided into the following categories:

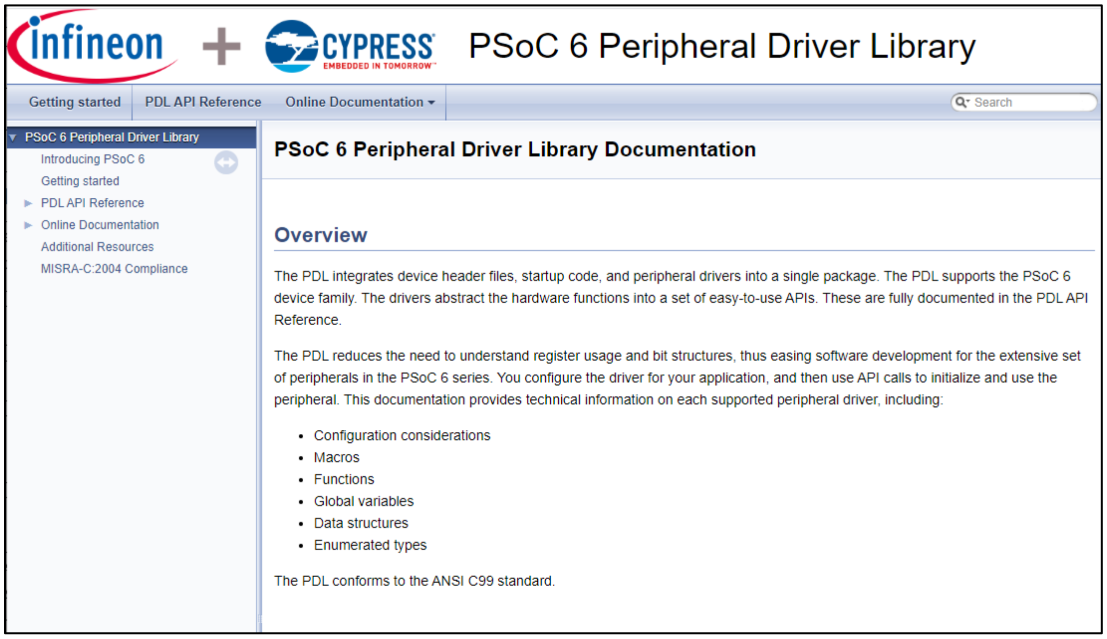
- A `_init` function allocates a block (along with any dependent resources), configures it, and enables it.
- A `_free` function disables a block, releases any dependent resources, and marks it as unused.
- Additional functions provide access to block-specific functionality and, in some cases, manipulate a block's configuration after it has been initialized.

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."


Quick Panel
(x)= Variables
Expressions
Breakpoints
USB Configurator 2.20 (new configuration)
Documentation
CY8CKIT-062S2-43012 BSP
Cypress CapSense Middleware Library 2.10: Cypress CapSense Middlew
Core Library (core-lib)
Hardware Abstraction Layer (HAL)
Overview
System View Description
PSoC 6 Peripheral Driver Library: Main Page



PSoC 6 Peripheral Driver Library Documentation

Overview

The PDL integrates device header files, startup code, and peripheral drivers into a single package. The PDL supports the PSoC 6 device family. The drivers abstract the hardware functions into a set of easy-to-use APIs. These are fully documented in the PDL API Reference.

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. You configure the driver for your application, and then use API calls to initialize and use the peripheral. This documentation provides technical information on each supported peripheral driver, including:

- Configuration considerations
- Macros
- Functions
- Global variables
- Data structures
- Enumerated types

The PDL conforms to the ANSI C99 standard.

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, libraries and library dependencies. 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. There is also a separate BSP dependencies manifest that lists the dependent libraries associated with each BSP.
- 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. There is also a separate middleware dependencies manifest that lists the dependent libraries associated with each middleware library.

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:/MyManifests/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. The following is a subset of the full set of manifest repos that exist. Each repo may also contain multiple manifests.

- Super Manifest: https://github.com/cypresssemiconductorco/mtb-super-manifest
- Code Example Manifest: https://github.com/cypresssemiconductorco/mtb-ce-manifest
- BSP Manifest: https://github.com/cypresssemiconductorco/mtb-bsp-manifest
- Middleware Manifest: https://github.com/cypresssemiconductorco/mtb-mw-manifest
- Wi-Fi Middleware Manifest: https://github.com/cypresssemiconductorco/mtb-wifi-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. Contact your IT department for the proxy name and port.

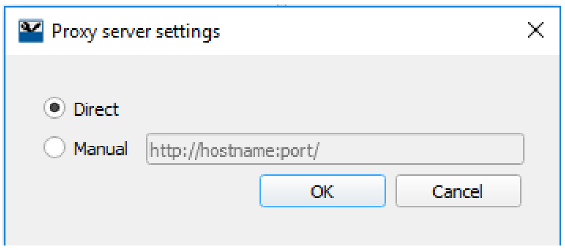
If you have environment variables for *http_proxy* and *https_proxy* (for example, to get Mbed to work), those same variables will work for ModusToolbox. However, there is a simpler way to enable/disable the proxy settings in the Project Creator and Library Manager tools that doesn't require you to create/delete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



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

We provide 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.

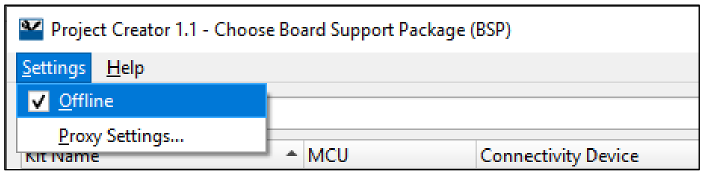
Extract the "offline" directory into the root location where you have ModusToolbox installed into a hidden directory named *.modustoolbox*. In most cases, this is in your User Home directory. After extracting, the path should be like this:

C:\Users\XYZ\.modustoolbox\offline

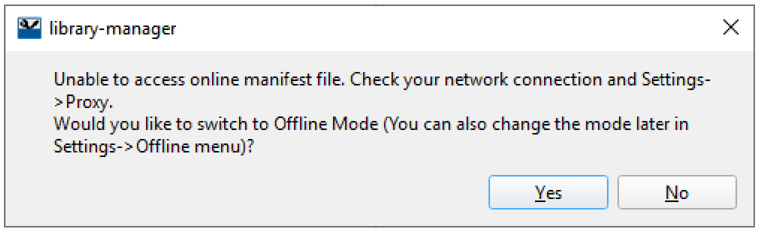
If you want to locate the offline content in a different directory, see the ModusToolbox Installation Guide.

Note If you have existing offline content, it is best to remove the entire offline directory before installing a new version.

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.


library-manager

Unable to access online manifest file. Check your network connection and Settings->Proxy.
Would you like to switch to Offline Mode (You can also change the mode later in Settings->Offline menu)?

5a.14 ModusToolbox 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

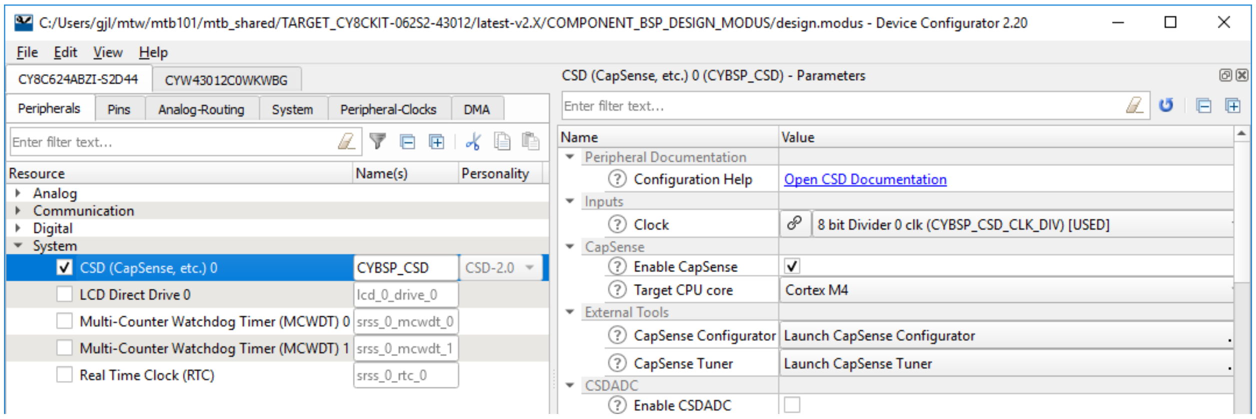
Configurators are divided into two types: Board Support Package (BSP) Configurators and Library Configurators.

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. As described earlier, since these files are often part of the BSP, if you use a configurator to modify them you will be modifying them for all applications that use the BSP (if it is shared). In addition, your changes will cause the BSP's repo to become dirty which means it cannot be updated to newer versions. Therefore, it is not recommended that you change any settings that are in the BSP. Instead, you can create a custom configuration for a single application (see Modifying the BSP Configuration (e.g. design.modus) for a Single Application) or you can create a custom BSP (see Creating your Own BSP).

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.


C:/Users/gjl/mtw/mtb101/mtb_shared/TARGET_CY8CKIT-062S2-43012/latest-v2.X/COMPONENT_BSP_DESIGN_MODUS/design.modus - Device Configurator 2.20
File Edit View Help
CY8C624ABZI-S2D44
CYW43012C0WKWBG
Peripherals Pins Analog-Routing System Peripheral-Clocks DMA
Enter filter text...
CSD (CapSense, etc.) 0 (CYBSP_CSD) - Parameters

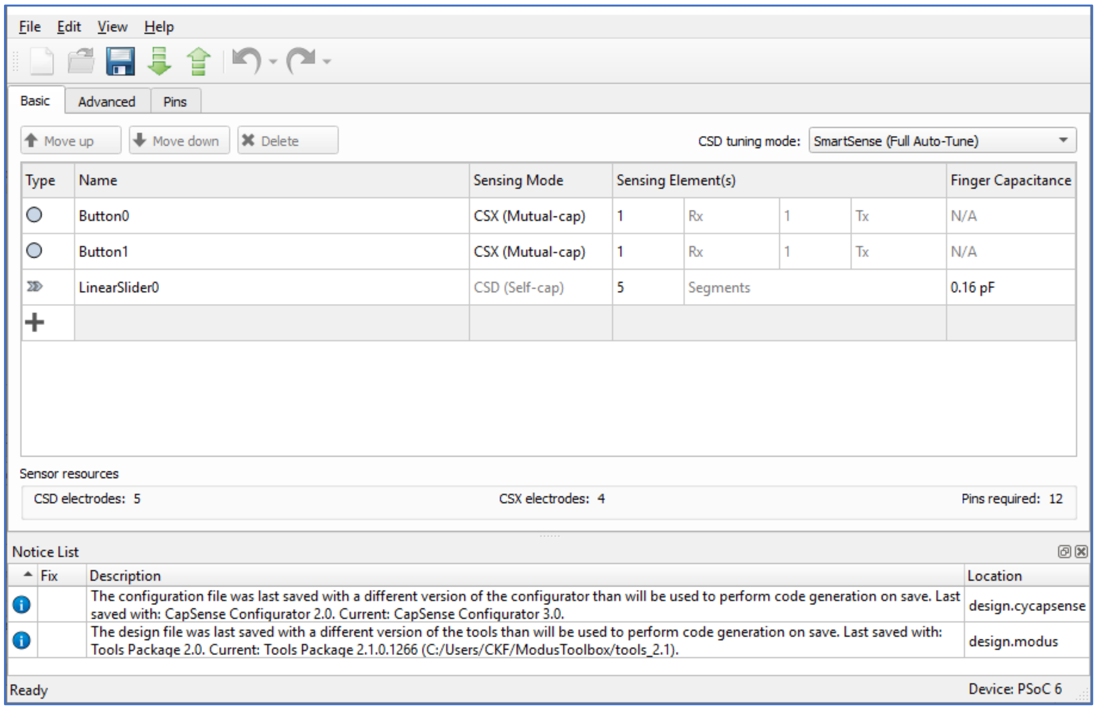
Resource	Name(s)	Personality
Analog		
Communication		
Digital		
System		
☑ CSD (CapSense, etc.) 0	CYBSP_CSD	CSD-2.0
☐ LCD Direct Drive 0	lcd_0_drive_0	
☐ Multi-Counter Watchdog Timer (MCWDT) 0	srss_0_mcwdt_0	
☐ Multi-Counter Watchdog Timer (MCWDT) 1	srss_0_mcwdt_1	
☐ Real Time Clock (RTC)	srss_0_rtc_0	

Name	Value
Peripheral Documentation	
Configuration Help	Open CSD Documentation
Inputs	
Clock	8 bit Divider 0 clk (CYBSP_CSD_CLK_DIV) [USED]
CapSense	
Enable CapSense	✓
Target CPU core	Cortex M4
External Tools	
CapSense Configurator	Launch CapSense Configurator
CapSense Tuner	Launch CapSense Tuner
CSDADC	
Enable CSDADC	☐

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.



CSD tuning mode: SmartSense (Full Auto-Tune)

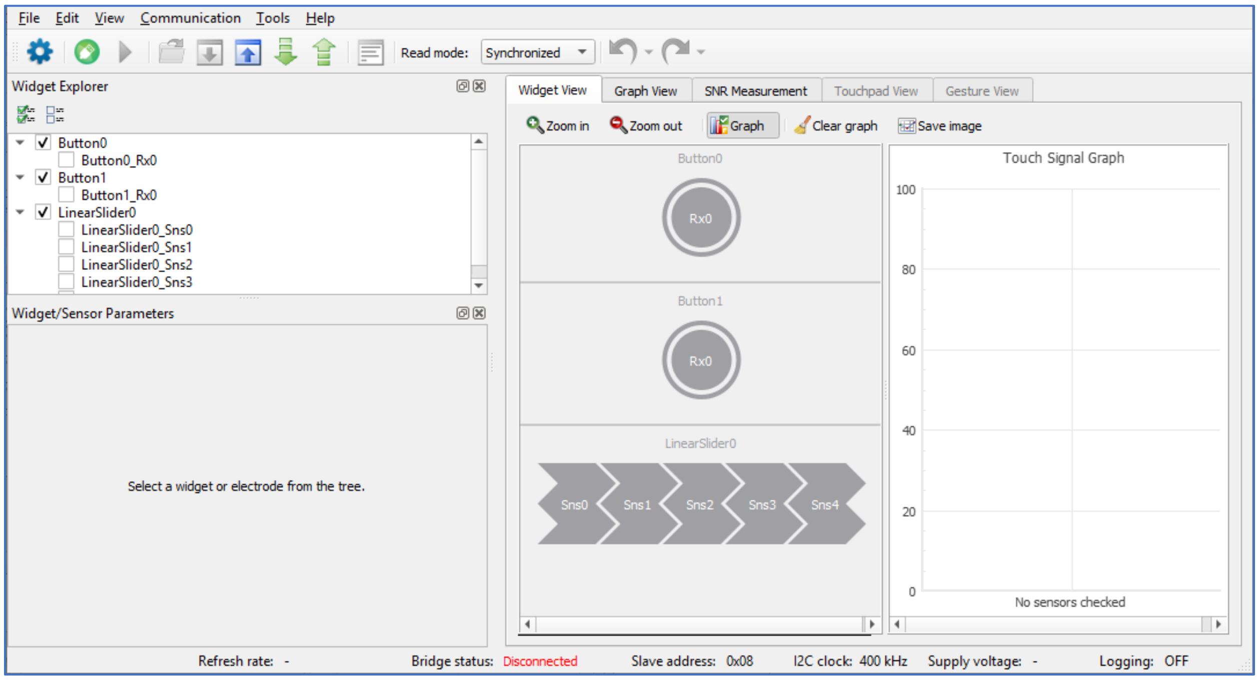
Type	Name	Sensing Mode	Sensing Element(s)				Finger Capacitance
○	Button0	CSX (Mutual-cap)	1	Rx	1	Tx	N/A
○	Button1	CSX (Mutual-cap)	1	Rx	1	Tx	N/A
	LinearSlider0	CSD (Self-cap)	5	Segments			0.16 pF
+							

Sensor resources
CSD electrodes: 5 CSX electrodes: 4 Pins required: 12

Notice List

Fix	Description	Location
	The configuration file was last saved with a different version of the configurator than will be used to perform code generation on save. Last saved with: CapSense Configurator 2.0. Current: CapSense Configurator 3.0.	design.cycapsense
	The design file was last saved with a different version of the tools than will be used to perform code generation on save. Last saved with: Tools Package 2.0. Current: Tools Package 2.1.0.1266 (C:/Users/CKF/ModusToolbox/tools_2.1).	design.modus

Ready Device: PSoC 6



Read mode: Synchronized

- ☑ Button0
  - ☐ Button0_Rx0
- ☑ Button1
  - ☐ Button1_Rx0
- ☑ LinearSlider0
  - ☐ LinearSlider0_Sns0
  - ☐ LinearSlider0_Sns1
  - ☐ LinearSlider0_Sns2
  - ☐ LinearSlider0_Sns3

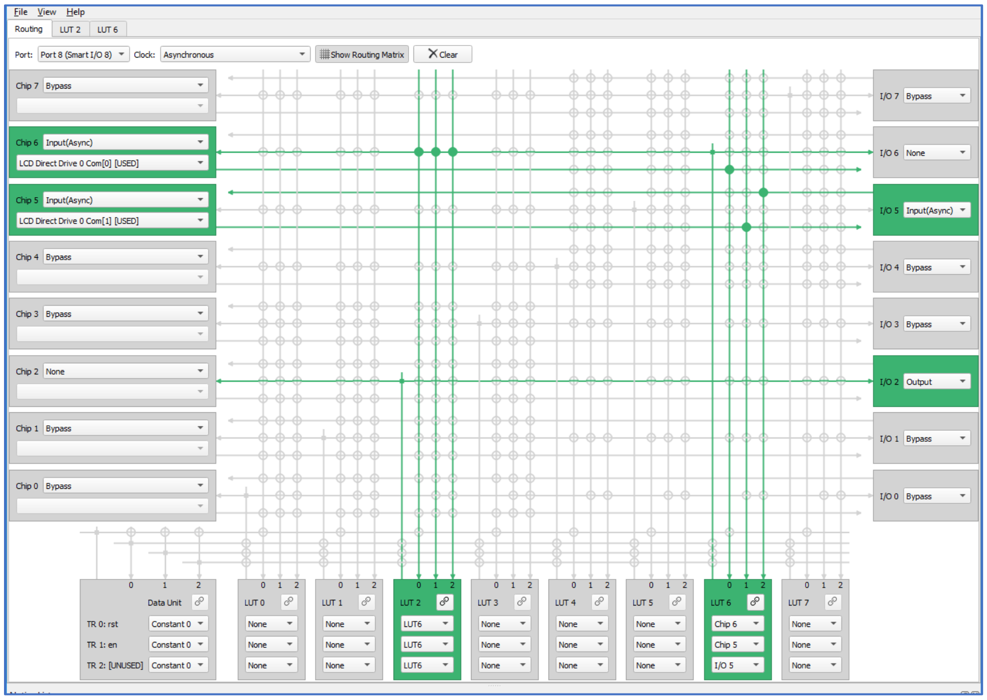
Select a widget or electrode from the tree.

No sensors checked

Refresh rate: - Bridge status: Disconnected Slave address: 0x08 I2C clock: 400 kHz Supply voltage: - Logging: OFF

Smart I/O™ Configurator

Configure the Smart I/O block, which adds programmable logic to an I/O port. There are usually only a few ports on a PSoC with Smart I/O capability, so it is important to consider that up front. On the PSoC from the CY8CKIT-062S2-43012, ports 8 and 9 have Smart I/O. Each port operates independently - that is, signals from port 8 cannot interact with signals from port 9.



Port: Port 8 (Smart I/O 8) Clock: Asynchronous

Chip 7	Bypass	
Chip 6	Input(Async)	LCD Direct Drive 0 Com[0] [USED]
Chip 5	Input(Async)	LCD Direct Drive 0 Com[1] [USED]
Chip 4	Bypass	
Chip 3	Bypass	
Chip 2	None	
Chip 1	Bypass	
Chip 0	Bypass	

I/O 7	Bypass
I/O 6	None
I/O 5	Input(Async)
I/O 4	Bypass
I/O 3	Bypass
I/O 2	Output
I/O 1	Bypass
I/O 0	Bypass

	Data Unit	LUT 0	LUT 1	LUT 2	LUT 3	LUT 4	LUT 5	LUT 6	LUT 7
TR 0: rst	Constant 0	None	None	LUT6	None	None	None	Chip 6	None
TR 1: en	Constant 0	None	None	LUT6	None	None	None	Chip 5	None
TR 2: [UNUSED]	Constant 0	None	None	LUT6	None	None	None	I/O 5	None

The port is chosen via a drop-down along the top of the window. Also selected at the top of the window is the clocking scheme. It can be asynchronous, or a variety of clock sources can be selected.

Along the left edge you will see a set of 8 chip connection points. These can be inputs to Smart I/O from chip peripheral outputs such as TCPWM signals or they can be outputs from Smart I/O to chip peripheral inputs such as SCB inputs.

The right edge allows connections to the chip's pins for that port - in our case, P8[7:0] and P9[7:0].

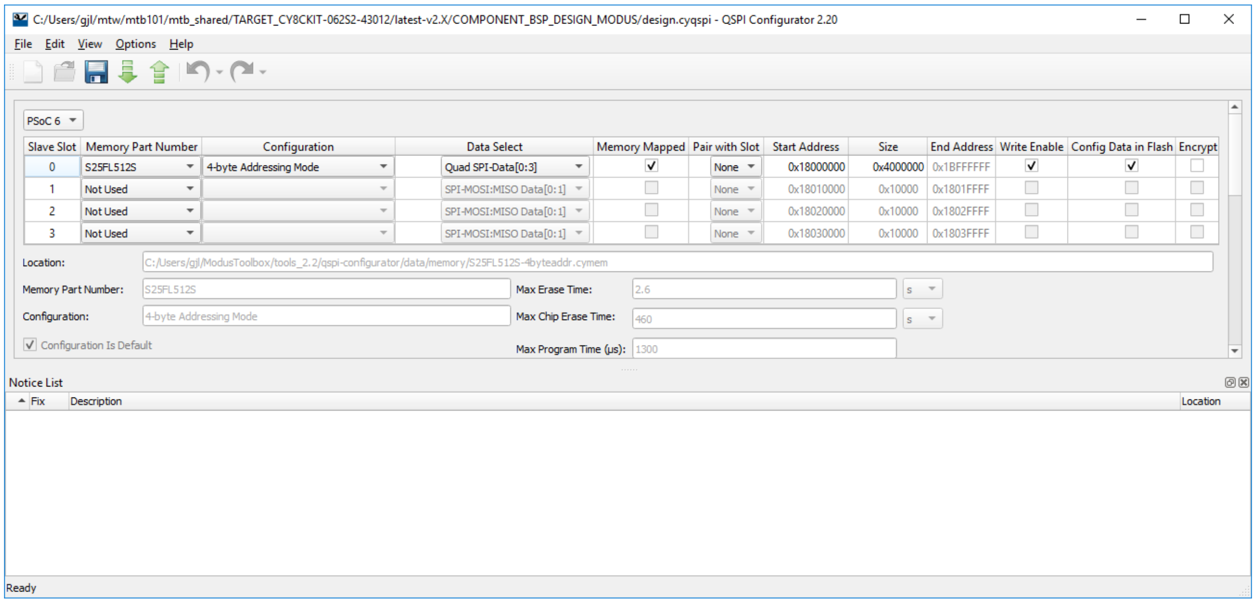
The bottom edge contains LUTs which can be used to combine and route signals to/from the chip's peripherals and to/from the chip's pins. The LUT inputs can come from peripherals, I/O pins, or from another LUT. Once you enable a LUT by selecting its inputs, a tab along the top appears for you to setup the truth table for the LUT. Note that if you don't need 3 inputs to a LUT, you can assign more than one input to the same signal.

The final section is the 8-bit Data Unit at the lower left corner. It allows you to setup trigger conditions that perform operations such as increment, decrement, and shift which are specified on the Data Unit tab once it is enabled by selecting one or more inputs.

Additional information is available in the Smart I/O user guide which can be found from the Help menu. You can also search for Smart I/O code examples in the usual places (In the Project Creator starter application list or on the Cypress GitHub site).

QSPI Configurator

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



C:/Users/gjl/mtw/mtb101/mtb_shared/TARGET_CY8CKIT-062S2-43012/latest-v2.X/COMPONENT_BSP_DESIGN_MODUS/design.cyqspi - QSPI Configurator 2.20

PSoC 6

Slave Slot	Memory Part Number	Configuration	Data Select	Memory Mapped	Pair with Slot	Start Address	Size	End Address	Write Enable	Config Data in Flash	Encrypt
0	S25FL512S	4-byte Addressing Mode	Quad SPI-Data[0:3]	✓	None	0x18000000	0x4000000	0x1BFFFFFF	✓	✓	☐
1	Not Used		SPI-MOSI:MISO Data[0:1]	☐	None	0x18010000	0x10000	0x1801FFFF	☐	☐	☐
2	Not Used		SPI-MOSI:MISO Data[0:1]	☐	None	0x18020000	0x10000	0x1802FFFF	☐	☐	☐
3	Not Used		SPI-MOSI:MISO Data[0:1]	☐	None	0x18030000	0x10000	0x1803FFFF	☐	☐	☐

Location: C:/Users/gjl/ModusToolbox/tools_2.2/qspi-configurator/data/memory/S25FL512S-4byteaddr.cymem

Memory Part Number: S25FL512S

Max Erase Time: 2.6 s

Configuration: 4-byte Addressing Mode

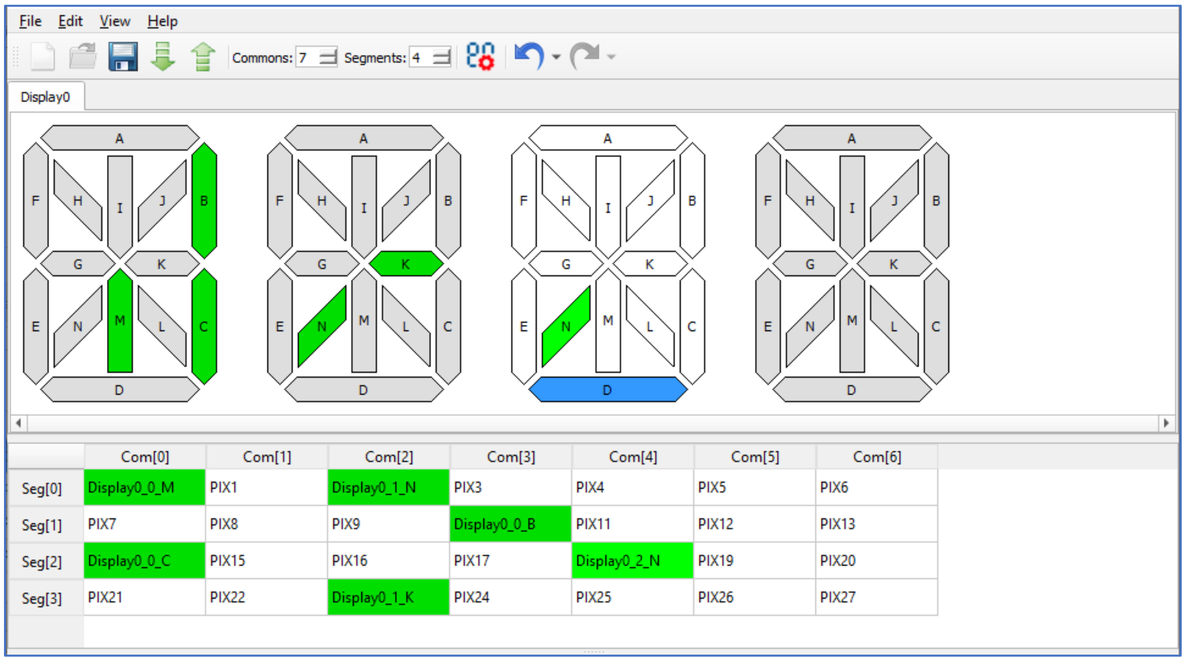
Max Chip Erase Time: 460 s

✓ Configuration Is Default

Max Program Time (µs): 1300

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.



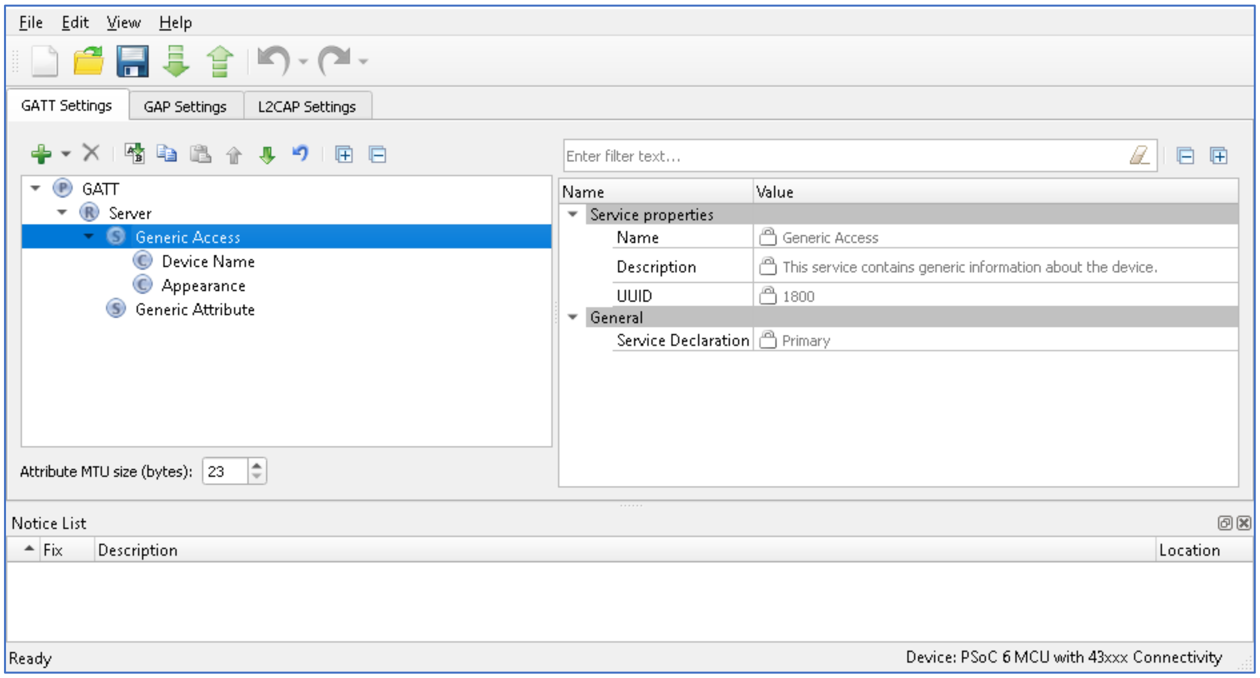
	Com[0]	Com[1]	Com[2]	Com[3]	Com[4]	Com[5]	Com[6]
Seg[0]	Display0_0_M	PIX1	Display0_1_N	PIX3	PIX4	PIX5	PIX6
Seg[1]	PIX7	PIX8	PIX9	Display0_0_B	PIX11	PIX12	PIX13
Seg[2]	Display0_0_C	PIX15	PIX16	PIX17	Display0_2_N	PIX19	PIX20
Seg[3]	PIX21	PIX22	Display0_1_K	PIX24	PIX25	PIX26	PIX27

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 GATT databases in generated code.



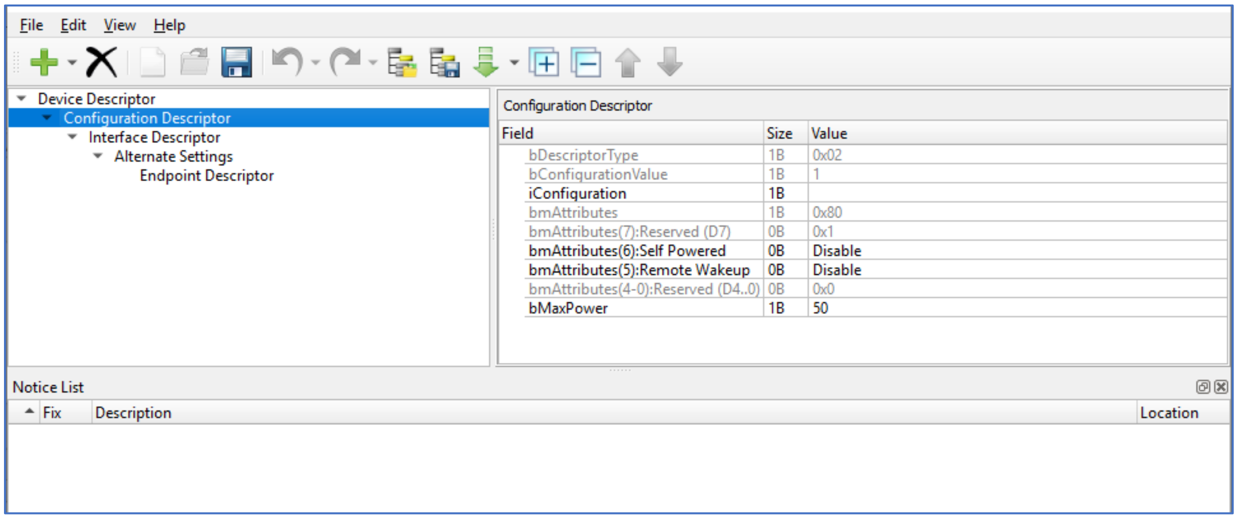
Attribute MTU size (bytes): 23

Name	Value
Service properties	
Name	Generic Access
Description	This service contains generic information about the device.
UUID	1800
General	
Service Declaration	Primary

Ready   Device: PSoC 6 MCU with 43xxx Connectivity

USB Configurator

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



Configuration Descriptor

Field	Size	Value
bDescriptorType	1B	0x02
bConfigurationValue	1B	1
iConfiguration	1B	
bmAttributes	1B	0x80
bmAttributes(7):Reserved (D7)	0B	0x1
bmAttributes(6):Self Powered	0B	Disable
bmAttributes(5):Remote Wakeup	0B	Disable
bmAttributes(4-0):Reserved (D4..0)	0B	0x0
bMaxPower	1B	50

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 *../mtb_shared/freertos/<version>/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 Compiler Optimization

The default compiler setting is "Debug". If you want to change this to "Release" to increase the optimization that is done, you must do 2 things:

1. Change the value of the variable `CONFIG` in the Makefile to "Release".
2. Regenerate or update any IDE launch configurations. This is necessary because the build output files go in a directory that has the `CONFIG` setting it the path. Therefore, if you don't update the configurations, the program and debug step will use the old build output files.

Note that the definition of Debug and Release are compiler dependent. For GCC_ARM, you can find the settings in *recipe-make-cat1a/<version>/make/toolchains/GCC_ARM.mk*:

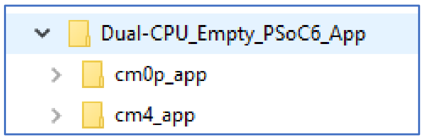
```
ifeq ($(CONFIG),Debug)
CY_TOOLCHAIN_DEBUG_FLAG=-DDEBUG
CY_TOOLCHAIN_OPTIMIZATION=-Og
else ifeq ($(CONFIG),Release)
CY_TOOLCHAIN_DEBUG_FLAG=-DNDEBUG
CY_TOOLCHAIN_OPTIMIZATION=-Os
else
CY_TOOLCHAIN_DEBUG_FLAG=
CY_TOOLCHAIN_OPTIMIZATION=
endif
```

You can also set the value of `CONFIG` to a custom name (e.g. "Custom") and then use the CFLAGS variable in the Makefile to set the optimization parameters as you wish.

5a.17 Dual CPU Designs

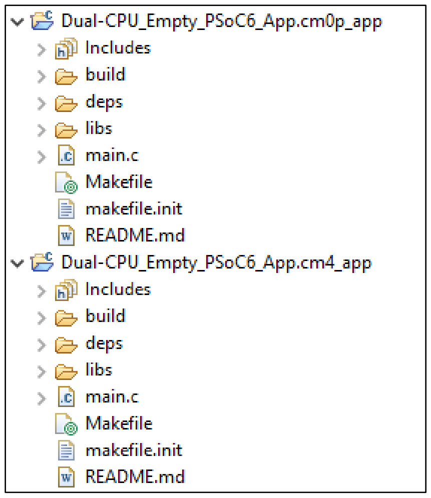
Many PSoC 6 devices contain 2 CPUs - a Cortex M4 and a Cortex M0+. Most code examples just use the M4 but the M0+ can also be used when necessary to offload the M4 or to achieve lower power. Since both CPUs use many of the same resources (AHB, Flash, Ram, etc.), it is critically important to consider how an application running on one CPU may affect the other.

Dual-CPU designs are accomplished using two separate projects that are bundled into a single application. When you create an application based on a dual-CPU design, on disk you will have a top-level application directory that contains the projects for each individual CPU. For example, the Dual-CPU Empty PSoC 6 application will look like this:


Dual-CPU_Empty_PSoC6_App
cm0p_app
cm4_app

Each application specifies its CY_GETLIBS_SHARED_PATH as *../../* (two levels up) instead of *../* (one level up). This accounts for the extra level of directory hierarchy so that the shared location is the same as for any other applications in the same workspace. (**Note** The above is true for Dual-CPU applications that use the MTB flow. Dual-CPU applications using the LIB flow include shared libraries using a different method.)

Inside the Eclipse IDE, each project will appear separately, but the hierarchy will be reflected in the project names:


Dual-CPU_Empty_PSoC6_App.cm0p_app
Includes
build
deps
libs
main.c
Makefile
makefile.init
README.md
Dual-CPU_Empty_PSoC6_App.cm4_app
Includes
build
deps
libs
main.c
Makefile
makefile.init
README.md

The Makefiles are set up so that if you build the CM4 project, it will build the CM0+ project as a dependency and will include it in the resulting build artifacts. Therefore, building/programming the CM4 project is all that is necessary to build and program both projects into the device.

See AN215656 (PSoC 6 Dual-CPU System Design) for much more information on Dual-CPU designs including inter-processor communication (IPC), interrupts and debugging.

Several Dual-CPU code examples are available in the usual places (In the Project Creator starter application list or on the Cypress GitHub site) to demonstrate using IPC for sharing data, semaphores for synchronization, etc.

5a.18 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.18.1 Exercise 3: Modify the blinking LED to use the Green LED

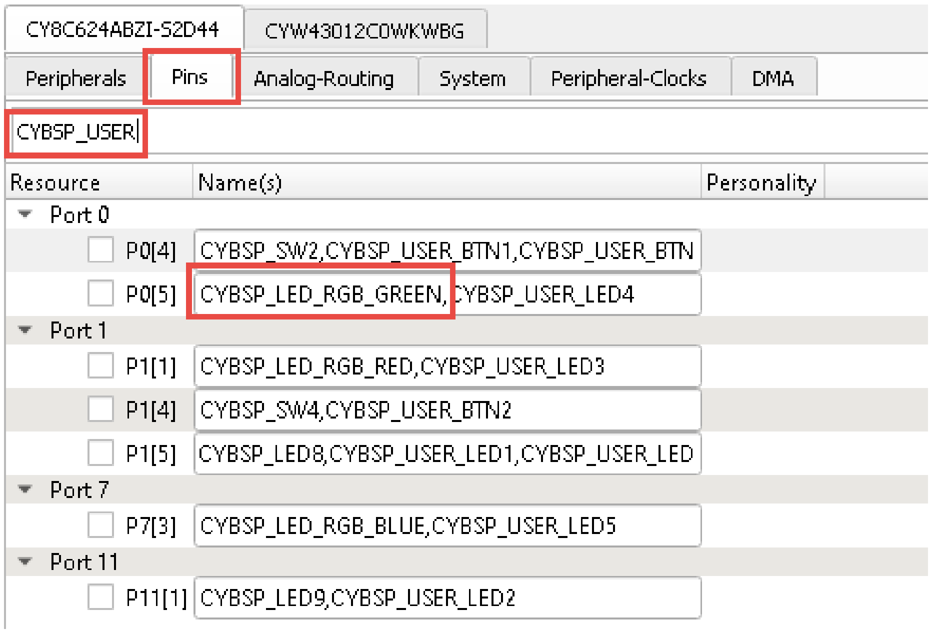
☐ 1. Open the *main.c* file from your previous project in an editor.

☐ 2. Locate this line of code:

```
cyhal_gpio_init(CYBSP_USER_LED1,CYHAL_GPIO_DIR_OUTPUT,CYHAL_GPIO_DRIV
E_STRONG, 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.

If you don't remember a pin name, they are defined in the Device Configurator Pins tab under the Name(s) column. User the filter text box at the top of the screen to filter the list.



Resource	Name(s)	Personality
Port 0		
P0[4]	CYBSP_SW2,CYBSP_USER_BTN1,CYBSP_USER_BTN	
P0[5]	CYBSP_LED_RGB_GREEN,CYBSP_USER_LED4	
Port 1		
P1[1]	CYBSP_LED_RGB_RED,CYBSP_USER_LED3	
P1[4]	CYBSP_SW4,CYBSP_USER_BTN2	
P1[5]	CYBSP_LED8,CYBSP_USER_LED1,CYBSP_USER_LED	
Port 7		
P7[3]	CYBSP_LED_RGB_BLUE,CYBSP_USER_LED5	
Port 11		
P11[1]	CYBSP_LED9,CYBSP_USER_LED2	

You can also find comments with the BSP pin names in the 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);
```

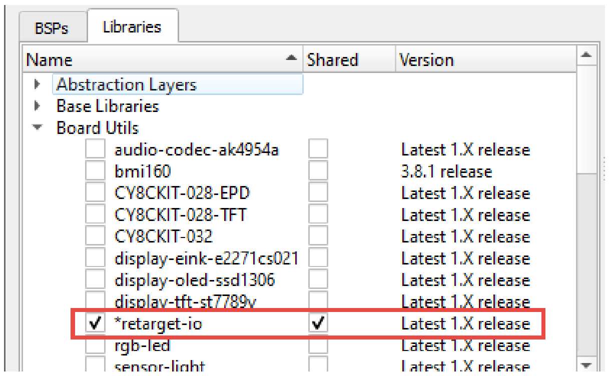
☐ 5. Build the project and program the board.

5a.18.2 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.18.3 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**.
2. Now let's use the Library Manager to add a library to our application.
   - In the IDE, you can launch the Library Manager 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`.
3. Switch to the **Libraries** tab.
4. Find the *retarget-io* library, check the box, and click **Update** to add the library to your application.



Name	Shared	Version
▸ Abstraction Layers		
▸ Base Libraries		
▾ Board Utils		
☐ audio-codec-ak4954a	☐	Latest 1.X release
☐ bmi160	☐	3.8.1 release
☐ CY8CKIT-028-EPD	☐	Latest 1.X release
☐ CY8CKIT-028-TFT	☐	Latest 1.X release
☐ CY8CKIT-032	☐	Latest 1.X release
☐ display-eink-e2271cs021	☐	Latest 1.X release
☐ display-oled-ssd1306	☐	Latest 1.X release
☐ display-tft-st7789v	☐	Latest 1.X release
✓ *retarget-io	✓	Latest 1.X release
☐ rgb-led	☐	Latest 1.X release
☐ sensor-light	☐	Latest 1.X release

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.

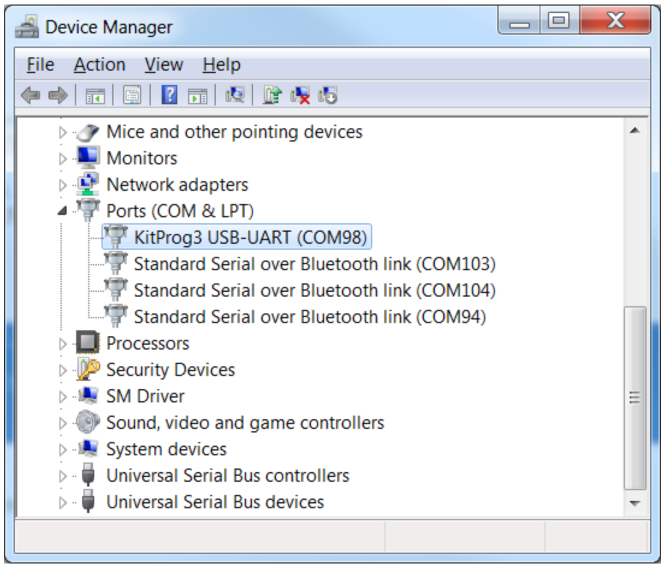
☐ 8. Before the forever loop, use `printf` to write “PSoC Rocks!\r\n” to the UART.

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.

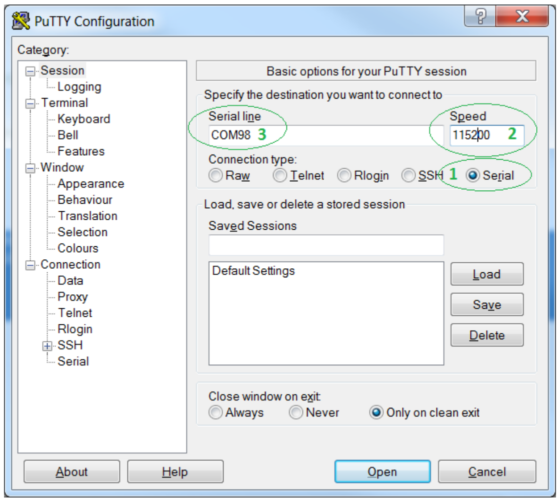
☐ 9. From the Windows Search, type “device manager” and launch that utility.

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


Device Manager
File Action View Help
Mice and other pointing devices
Monitors
Network adapters
Ports (COM & LPT)
KitProg3 USB-UART (COM98)
Standard Serial over Bluetooth link (COM103)
Standard Serial over Bluetooth link (COM104)
Standard Serial over Bluetooth link (COM94)
Processors
Security Devices
SM Driver
Sound, video and game controllers
System devices
Universal Serial Bus controllers
Universal Serial Bus devices

☐ 11. From Windows, launch the PuTTY application (or your favorite terminal emulator).

12. Specify a Serial session, at a Speed of 115200 (baud rate), using the Serial Line from the device Manager (COM PORT).



Serial line: COM98
Speed: 115200
Connection type: Raw Telnet Rlogin SSH [✓ Serial]
Close window on exit: Always Never [✓ Only on clean exit]

13. Build the project and program the board. Observe the on the terminal window when programming completes.

5a.18.4 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_on(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.18.5 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

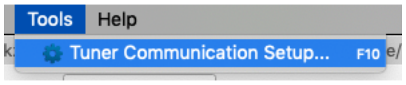
☐ 1. Launch the CapSense Tuner through the IDE or in stand-alone mode. Note that, in the latter case, you will need to manually open the CapSense configuration file:

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:

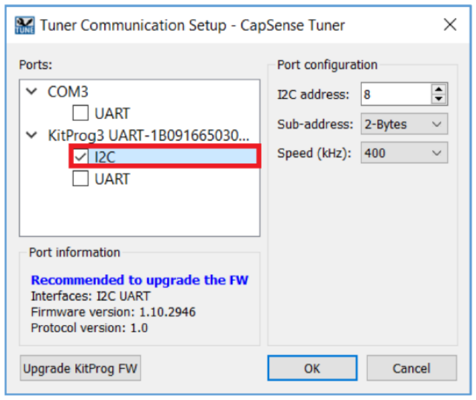
```
make open CY_OPEN_TYPE=capsense-tuner
```

☐ 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:


Tuner Communication Setup - CapSense Tuner
Ports:
COM3
UART
KitProg3 UART-1B091665030...
I2C
UART
Upgrade KitProg FW
OK
Cancel

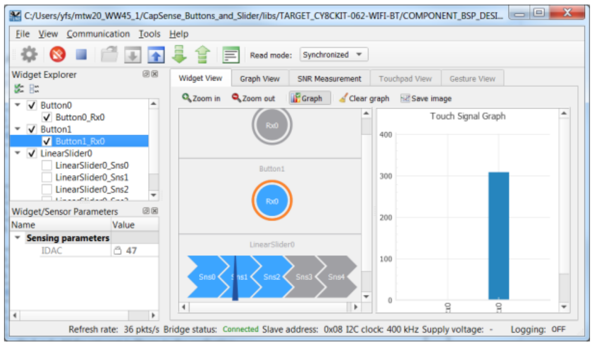
Ports: KitProg3 UART-1B091665030... [✓ I2C]

Port configuration
I2C address: 8
Sub-address: 2-Bytes
Speed (kHz): 400

Port information
Recommended to upgrade the FW
Interfaces: I2C UART
Firmware version: 1.10.2946
Protocol version: 1.0

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.



Refresh rate: 36 pkts/s Bridge status: Connected Slave address: 0x08 I2C clock: 400 kHz Supply voltage: - Logging: OFF

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

5a.18.6 Exercise 8: Write a message to the TFT shield

This project displays a message on the CY8CKIT-028-TFT shield when it 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 **Update**. Once updates are done, close the Library Manager.
4. Open the project Makefile to add emwin configuration for bare metal without touchscreen:

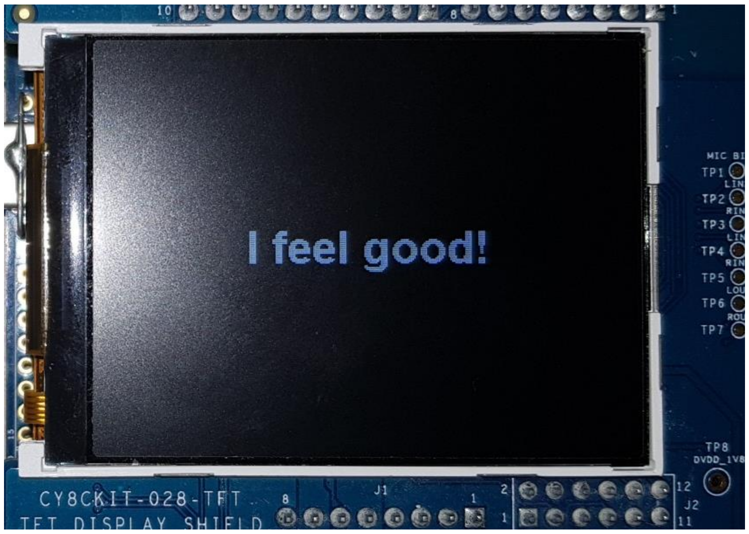
```
COMPONENTS+=EMWIN_NOSNTS
```

☐ 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"
#include "cy8ckit_028_tft.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();

    /* Initialize the CY8CKIT_028_TFT board */
    cy8ckit_028_tft_init (NULL, NULL, NULL, NULL);

    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.18.7 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 *../mtb_shared/freertos/<version>/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. Make sure to add a function declaration if you define the function after main.

```
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);
    }
}
```

Note Do not use `Cy_SysLib_Delay` in a task because it will not do what you want - use `vTaskDelay` instead.

☐ 8. In `main` after the initialization code, 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 and update it to use LED2. Change the delay value so the LEDs blink asynchronously.

☐ 11. In `main` create this task with the same priority and stack size.

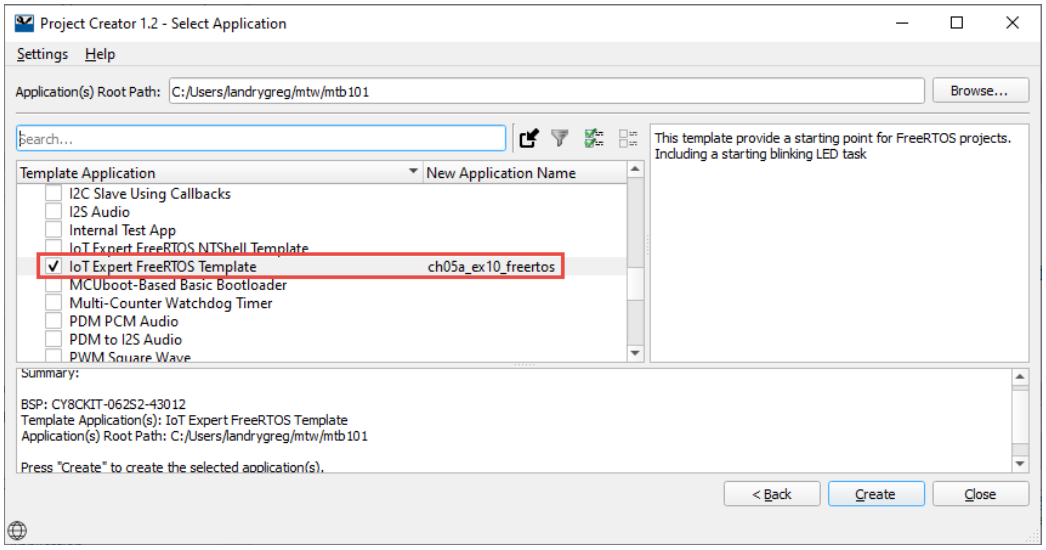
☐ 12. Build the program and program the kit. Observe the LEDs blinking at different rates.

5a.18.8 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 "IoT Expert FreeRTOS Template" that gets picked up by the Project Creator from the iotexpert manifests. Name the project **ch05a_ex10_freertos**.



Application(s) Root Path: C:/Users/landrygreg/mtw/mtb101

This template provide a starting point for FreeRTOS projects. Including a starting blinking LED task

Template Application	New Application Name
I2C Slave Using Callbacks	
I2S Audio	
Internal Test App	
IoT Expert FreeRTOS NTShell Template	
✓ IoT Expert FreeRTOS Template	ch05a_ex10_freertos
MCUboot-Based Basic Bootloader	
Multi-Counter Watchdog Timer	
PDM PCM Audio	
PDM to I2S Audio	
PWM Square Wave	

Summary:

BSP: CY8CKIT-062S2-43012
Template Application(s): IoT Expert FreeRTOS Template
Application(s) Root Path: C:/Users/landrygreg/mtw/mtb101

Press "Create" to create the selected application(s).

☐ 3. Build and Program.

☐ 4. What does the project do?

☐ 5. Does this application use the MTB flow or the LIB flow?

6. Make the LED blink faster.
7. Add the ntshell library.

   It is in the iotexpert GitHub site. It will show up in the Library Manager under the category **IoT Expert** because of the manifest files that we added.

8. Follow the instructions in the mtb_shared/middleware-ntshell/<version>/README.md file to create the ntshell task.

   Look in the section entitled "Get the source templates".

9. Build and Program.
10. Open a terminal emulator to test the shell. Type `help` and press enter 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.

    Look in the README.md file in the section "Adding Commands".

    One suggested implementation:

    1. Create a global variable for the led rate in usrcmd.c and initialize its value. Add your variable as an extern in usrcmd.h.
    2. Include <stdio.h> in usrcmd.h to get variable type definitions.
    3. When implementing the function, you can use `atoi` to convert the first argument (`argv[1]`) to an integer. You will need to include <stdlib.h> to use `atoi`.
    4. Use your variable in main.c as the value for vTaskDelay.

5a.18.9 Exercise 11: Create a Custom Device Configuration

In this exercise, you will copy the configuration files from the BSP to the application and will over-ride the configuration for that single application. This is useful if you want to set custom pin, peripheral, or clock settings for an application using the Device Configurator rather than the HAL. This is most commonly done for features or settings not supported directly by the HAL.

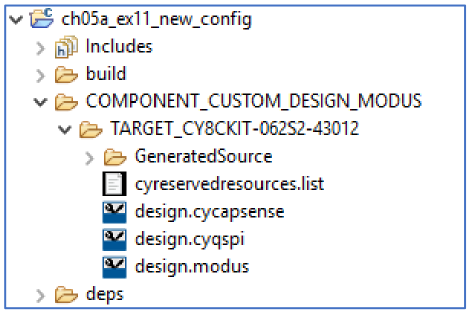
1. Run the project creator tool to create a new Hello World application for your kit. Name it **ch05a_ex11_custom_config**.
2. Build and program the kit and observe the LED that pauses/resumes blinking when you press the return key in your terminal emulator.

```
make program
```

3. Create a directory and sub-directory inside the root folder of your application as follows:

```
COMPONENT_CUSTOM_DESIGN_MODUS/TARGET_CY8CKIT-062S2-43012
```

4. Copy the contents of *mtb_shared/TARGET_CY8CKIT-062S2-43012/<version>/COMPONENT_BSP_DESIGN_MODUS* to the subdirectory you created. It should look like this when you are done:



```
ch05a_ex11_new_config
  Includes
  build
  COMPONENT_CUSTOM_DESIGN_MODUS
    TARGET_CY8CKIT-062S2-43012
      GeneratedSource
      cyreservedresources.list
      design.cycapsense
      design.cyqspi
      design.modus
  deps
```

☐ 5. Edit the Makefile to disable the configuration from the BSP and enable your custom configuration:

Add *CUSTOM_DESIGN_MODUS* to the *COMPONENTS* variable.

```
COMPONENTS+=CUSTOM_DESIGN_MODUS
```

Add *BSP_DESIGN_MODUS* to the *DISABLE_COMPONENTS* variable.

```
DISABLE_COMPONENTS+=BSP_DESIGN_MODUS
```

☐ 6. Run the device configurator.

```
make config
```

If you are using the Eclipse IDE, don't forget to refresh the Quick Panel first to get the correct file.

In either case (command line or IDE) it is worth looking at the banner in the configurator to be sure the correct file is opened.

☐ 7. Move the definition for `CYBSP_USER_LED` to the pin that has `CYBSP_USER_LED2`. Save and close the device configurator when done.

Hint: Enter LED into the search box to see only pins with LED in their name.
Hint: Do not use any spaces between names (just a comma) since spaces will be converted to underscores.

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

5a.18.10 Exercise 12: Create and Use a Custom BSP

In this exercise, we will do the same thing but with a complete new custom BSP to use. This is useful if you are building your own hardware and need to make more changes than just the configuration.

☐ 1. Run the project creator tool to create a new Hello World application for your kit. Name it **ch05a_ex12_custom_bsp**.

☐ 2. 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
```

☐ 3. Now create a new BSP called “MYKIT”. We won't specify a different PSoC 6 or connectivity device in this case. From the command line:

```
make bsp TARGET_GEN=MYKIT
```

☐ 4. Edit the Makefile to use to specify the new BSP as the target.

```
TARGET=MYKIT
```

☐ 5. Open the Device Configurator.

```
make config
```

If you are using the Eclipse IDE to open the configurator, don’t forget to refresh the Quick Panel first to get the correct file.

In either case (command line or IDE) it is worth looking at the banner in the configurator to be sure the correct file is opened.

☐ 6. Move the definition for `CYBSP_USER_LED` to the pin that has `CYBSP_USER_LED2`. Save and close the device configurator when done.

Hint: Enter “LED” in the text filter box to see only pins with LED in their name.
Hint: Put a comma between names but do NOT put any spaces in the Name(s) field since they will be converted to underscores.

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

```
make program
```