

PSoC™ 6 Digital Documentation

Table of contents

	Table of contents	1
1	PSoC™ 6 MCU overview	26
2	PSoC™ 6 datasheets	29
3	PSoC™ 6 software tools	30
3.1	ModusToolbox™ installation guide	30
3.1	About this document	30
3.1.1	General information	31
3.1.1.1	System requirements	31
3.1.1.2	Uninstall Beta versions	31
3.1.2	Step 1: Download the software	32
3.1.2.1	ModusToolbox™ patches	32
3.1.2.2	Prerequisites	32
3.1.2.3	SEGGER J-Link	32
3.1.3	Step 2: Install ModusToolbox™ software	33
3.1.3.1	Installing in non-default location	33
3.1.3.2	Installing with previous versions	33
3.1.3.3	Windows	33
3.1.3.4	Linux	34
3.1.3.5	macOS	34
3.1.4	Step 3: Run the Eclipse IDE	35
3.1.5	Default versus advanced Windows installation	36
3.1.6	Installing with spaces in user home directory	37
3.1.6.1	Step 1: Install at a custom path.....	37
3.1.6.2	Step 2: Create a variable to specify the path to Tools.....	38
3.1.6.3	Step 3: Create a variable to specify the path to cache.....	38
3.1.6.4	Step 4: Specify the custom path to use for offline content and manifest.loc.....	39
3.1.6.4.1	Offline content path	39
3.1.6.4.2	manifest.loc	39
3.1.7	Revision history	40
3.2	Eclipse IDE for ModusToolbox™	40
3.2	About this document	40
3.2.1	Download/Install/Run Eclipse IDE	42
3.2.2	Create new application	43
3.2.2.1	Step 1: Open Project Creator tool	43
3.2.2.2	Step 2: Choose Board Support Package (BSP)	44
3.2.2.3	Step 3: Select application	45

Table of contents

3.2.2.4	Step 4: Create application	45
3.2.3	Add/modify application code	47
3.2.4	View device resources	48
3.2.5	Build the Application	49
3.2.6	Program the Device	50
3.2.6.1	Connect the Kit	50
3.2.6.2	Program	50
3.2.7	Debug the program	52
3.2.8	Revision history	54
3.3	ModusToolbox™ user guide	54
3.3	About this document	54
3.3.1	Introduction	55
3.3.1.1	What is ModusToolbox™ software?	55
3.3.1.2	Run-time software	55
3.3.1.2.1	Code examples	56
3.3.1.2.2	Libraries (middleware)	56
3.3.1.2.3	BSPs	57
3.3.1.3	Development tools	58
3.3.1.3.1	Directory structure	58
3.3.1.3.2	Documentation	59
3.3.1.3.3	IDE support	60
3.3.1.3.4	Tools	60
3.3.1.4	Product versioning	64
3.3.1.4.1	General philosophy	64
3.3.1.4.2	Tools package versioning	65
3.3.1.4.3	Multiple tools versions installed	65
3.3.1.4.4	Specifying alternate tools version	66
3.3.1.4.5	Tools and configurators versioning	67
3.3.1.4.6	GitHub libraries versioning	68
3.3.1.4.7	Dependencies between libraries	68
3.3.1.5	Partner ecosystems	69
3.3.2	Getting started	70
3.3.2.1	Install and configure software	70
3.3.2.1.1	GUI set-up instructions	70
3.3.2.1.2	CLI set-up instructions	70
3.3.2.2	Get help	70
3.3.2.2.1	GUI Documentation	71
3.3.2.2.2	Command line documentation	71
3.3.2.3	Create applications	72
3.3.2.3.1	Project Creator tools	72
3.3.2.3.2	git clone	74
3.3.2.3.3	Typical application contents	74

Table of contents

3.3.2.4	Update BSPs and libraries	76
3.3.2.4.1	Library Manager	76
3.3.2.4.2	make getlibs	77
3.3.2.5	Configure settings for devices, peripherals, and libraries	78
3.3.2.5.1	Configurator GUI tools	78
3.3.2.5.2	Configurator CLI tools	79
3.3.2.6	Write application code	79
3.3.2.6.1	Application layers	80
3.3.2.7	Build, program, and debug	81
3.3.2.7.1	Use Eclipse IDE	82
3.3.2.7.2	Export to another IDE	82
3.3.2.7.3	Use command line	82
3.3.3	ModusToolbox™ build system	84
3.3.3.1	Overview	84
3.3.3.2	Application types	84
3.3.3.3	BSPs	85
3.3.3.4	make getlibs	85
3.3.3.4.1	repos	85
3.3.3.5	Adding source files	86
3.3.3.5.1	Auto-discovery	86
3.3.3.6	Pre-builds and post-builds	88
3.3.3.7	Program and debug	89
3.3.3.8	Available make targets	89
3.3.3.8.1	General make targets	89
3.3.3.8.2	IDE make targets	90
3.3.3.8.3	Tools make targets	90
3.3.3.8.4	Utility make targets	91
3.3.3.9	Available make variables	92
3.3.3.9.1	Basic configuration make variables	92
3.3.3.9.2	Advanced configuration make variables	93
3.3.3.9.3	BSP make variables	95
3.3.3.9.4	Getlibs make variables	95
3.3.3.9.5	Path make variables	97
3.3.3.9.6	Miscellaneous make variables	98
3.3.4	Board support packages	100
3.3.4.1	Overview	100
3.3.4.2	What's in a BSP	100
3.3.4.2.1	COMPONENT_BSP_DESIGN_MODUS	102
3.3.4.2.2	COMPONENT	102
3.3.4.2.3	deps subdirectory	102
3.3.4.2.4	docs subdirectory	102
3.3.4.2.5	Support files	103

Table of contents

3.3.4.2.6	<BSP_NAME>.mk	103
3.3.4.2.7	locate_recipe.mk	103
3.3.4.2.8	README/RELEASE.md	103
3.3.4.2.9	BTSDK-specific BSP files	103
3.3.4.3	Creating your own BSP	103
3.3.4.4	Modifying the BSP configuration for a single application	105
3.3.5	Manifest files	107
3.3.5.1	Overview	107
3.3.5.2	Create your own manifest	108
3.3.5.2.1	Overriding the standard super-manifest	109
3.3.5.2.2	Secondary super-manifest	109
3.3.5.2.3	Processing	109
3.3.5.2.4	Conflicting data	110
3.3.5.3	Using offline content	110
3.3.5.4	Access private repositories	111
3.3.6	Using applications with third-party tools	113
3.3.6.1	Import to Eclipse	113
3.3.6.2	Exporting to supported IDEs	114
3.3.6.2.1	Overview	114
3.3.6.2.2	Export to VS Code	114
3.3.6.2.3	Export IAR EWARM (Windows only)	118
3.3.6.2.4	Export to Keil µVision 5 (Windows only)	124
3.3.6.3	Patched flashloaders for AIROC™ CYW208xx devices	141
3.3.6.4	Generating files for XMC™ Simulator tool	141
3.3.6	Revision history	142
4	PSoC™ 6 code examples	142
5	PSoC™ 6 application notes	154
5.1	AN228571 Getting started with PSoC™ 6 MCU on ModusToolbox™ software	154
5.1	About this document	154
5.1.1	Introduction	155
5.1.2	Development ecosystem	158
5.1.2.1	PSoC™ resources	158
5.1.2.2	Firmware/application development	158
5.1.2.2.1	Installing the ModusToolbox™ tools package	159
5.1.2.2.2	Choosing an IDE	159
5.1.2.2.3	ModusToolbox™ software	159
5.1.2.2.4	ModusToolbox™ applications	162
5.1.2.2.5	PSoC™ 6 software resources	165
5.1.2.2.6	ModusToolbox™ help	167
5.1.2.3	Support for other IDEs	168
5.1.2.4	FreeRTOS support with ModusToolbox™	168

Table of contents

5.1.2.5	Programming/debugging using Eclipse IDE	169
5.1.2.6	PSoC™ 6 MCU development kits	170
5.1.3	Device features	171
5.1.4	My first PSoC™ 6 MCU design using Eclipse IDE for ModusToolbox™ software	173
5.1.4.1	Prerequisites	173
5.1.4.1.1	Hardware	173
5.1.4.1.2	Software	173
5.1.4.2	Using these instructions	173
5.1.4.3	About the design	174
5.1.4.4	Part 1: Create a new application	174
5.1.4.5	Part 2: View and modify the design configuration	177
5.1.4.5.1	Opening the Device Configurator	179
5.1.4.5.2	Add retarget-io middleware	180
5.1.4.5.3	Configuration of UART, timer peripherals, pins, and system clocks	181
5.1.4.6	Part 3: Write firmware	182
5.1.4.7	Part 4: Build the application	189
5.1.4.8	Part 5: Program the device	189
5.1.4.9	Part 6: Test your design	192
5.1.5	Summary	195
5.1	References	196
5.1	Glossary	197
5.1.6	Revision history	198
5.2	AN221774 Getting started with PSoC™ 6 MCU on PSoC™ Creator	198
5.2	About this document	198
5.2.1	Introduction	200
5.2.1.1	Prerequisites	201
5.2.1.1.1	Hardware	201
5.2.1.1.2	Software	201
5.2.2	Development ecosystem	202
5.2.2.1	PSoC™ resources	202
5.2.2.2	Firmware/application development	202
5.2.2.2.1	Choosing an IDE	203
5.2.2.2.2	PSoC™ Creator	203
5.2.2.2.3	Software development kits for PSoC™ 6 devices	204
5.2.2.3	Support for other IDEs	205
5.2.2.4	RTOS support	206
5.2.2.4.1	RTOS support with PSoC™ Creator	206
5.2.2.5	Programming/debugging	206
5.2.2.6	PSoC™ 6 MCU development kits	207
5.2.3	Device features	208
5.2.4	My first PSoC™ 6 MCU design using PSoC™ Creator	210
5.2.4.1	Using these instructions	210

Table of contents

5.2.4.2	About the design	210
5.2.4.3	Part 1: Create a new project from scratch	211
5.2.4.4	Part 2: Implement the design	215
5.2.4.5	Part 3: Generate source code	224
5.2.4.6	Part 4: Write the firmware	226
5.2.4.7	Part 5: Build the project and program the device	231
5.2.4.8	Part 6: Test your design	233
5.2.5	Summary	235
	Related application notes and code examples	235
5.2	Glossary	237
5.2.6	Revision history	238
5.3	AN210781 Getting started with PSoC™ 6 MCU with Bluetooth® low energy connectivity on PSoC™ Creator	238
5.3	About this document	238
5.3.1	Introduction	239
5.3.1.1	Prerequisites	241
5.3.1.1.1	Hardware	241
5.3.1.1.2	Software	241
5.3.2	Development ecosystem	242
5.3.2.1	PSoC™ resources	242
5.3.2.2	Firmware/application development	242
5.3.2.2.1	PSoC™ Creator	242
5.3.2.2.2	Peripheral Driver Library (PDL)	243
5.3.2.3	Support for other IDEs	244
5.3.2.3.1	Using PSoC™ Creator to target another IDE	244
5.3.2.4	RTOS support	247
5.3.2.5	Debugging	247
5.3.2.6	CY8CKIT-062-BLE PSoC™ 6-BLE pioneer kit	248
5.3.2.7	CySmart Host Emulation Tool and mobile applications	248
5.3.3	Device features	249
5.3.4	Development setup	251
5.3.5	My first PSoC™ 6 MCU design with Bluetooth® LE	253
5.3.5.1	Using the instructions	253
5.3.5.2	Before you begin	253
5.3.5.3	About the design	254
5.3.5.4	Part 1. Create a new project from scratch	254
5.3.5.5	Part 2. Implement the design	260
5.3.5.6	Part 3. Generate source code	277
5.3.5.7	Part 4. Write the firmware	279
5.3.5.8	Part 5. Build the project, program the Device	290
5.3.5.9	Part 6. Test your design	293
5.3.6	Summary	299

Table of contents

5.3.7	Related application notes and code examples	300
A	Appendix A. Glossary	302
B	Appendix B. Bluetooth® LE protocol	303
B.1	Overview	303
B.2	Physical Layer (PHY)	303
B.3	Link Layer (LL)	303
B.4	Host Control Interface (HCI)	304
B.5	Logical Link Control and Adaptation protocol (L2CAP)	304
B.6	Security manager (SM)	305
B.7	Attribute protocol (ATT)	305
B.7.1	Attribute hierarchy	306
B.7.2	Attribute operations	308
B.8	Generic Attribute Profile (GATT)	308
B.9	Generic Access Profile (GAP)	309
C	Appendix C. Device features	312
C.1	System wide resources	312
C.1.1	CPU subsystem: CM4 and CM0	312
C.1.2	IPC	312
C.1.3	Memory system	312
C.1.4	DMA	312
C.1.5	Clocking system	313
C.1.6	System interrupts	313
C.1.7	Power supply and monitoring	313
C.1.8	Power modes	314
C.2	Secure Boot	315
C.3	Programmable digital peripherals	315
C.3.1	UDB	315
C.3.2	Programmable TCPWM	316
C.3.3	SCB	316
C.3.4	BLESS	316
C.3.5	Audio subsystem	317
C.3.6	Serial Memory Interface	318
C.3.7	eFUSE	318
C.3.8	Segment LCD	318
C.4	Programmable analog peripherals	318
C.4.1	Continuous Time Block Opamps	318
C.4.2	Low-Power comparator	318
C.4.3	SAR ADC	318
C.4.4	DAC	319
C.4.5	CAPSENSE™	319
C.5	Programmable GPIOs	319
D	Appendix D. IoT development tools	321

Table of contents

D.1	CY8CKIT-062-BLE PSoC™ 6-BLE pioneer kit	321
D.2	CySmart Host Emulation Tool	321
D.3	CySmart mobile app	323
5.3.12	Revision history	325
5.4	AN228753 PSoC™ 6 MCU usage of Direct Memory Access (DMA)	325
5.4	About this document	325
5.4.1	Introduction	326
5.4.2	Architecture	327
5.4.2.1	Transfer modes	328
5.4.3	DMA design	329
5.4.3.1	Step 1: Choose the DMA channel	329
5.4.3.2	Step 2: Configure triggers	329
5.4.3.3	Step 3: Set up the DMA channel	330
5.4.3.4	Step 4: Set up the DMA descriptor	330
5.4.3.5	Step 5: Write the user code	330
5.4.4	Priorities and preemption	332
5.4.5	Data transfer widths	334
5.4.6	Types of transfers	335
5.4.6.1	1-to-1 transfer	335
5.4.6.2	1-to-N transfer	336
5.4.6.2.1	Noncontiguous source/destination increments	337
5.4.6.3	N-to-1 transfer	338
5.4.6.4	N-to-N transfer	338
5.4.6.5	N-to-NxM	339
5.4.7	Chaining descriptors	341
5.4.8	Chaining DMA channels	343
5.4.9	Differences between DMA (DW) and DMAC	344
5.4.10	DMA transfer performance	345
5.4.10.1	Elements of a transfer	345
5.4.10.2	DMA (DW) and DMAC: trigger schemes and performance	348
5.4.10.3	Preemption and its impact on performance	349
5.4.10.4	Bus arbitration and its impact	350
5.4.11	Summary	351
5.4	References	352
5.4.12	Revision history	353
5.5	AN233648 PSoC™ 6 MCU: Modify CY8CPROTO-062-4343W board to work with an external flash memory	353
5.5	About this document	353
5.5.1	Introduction	354
5.5.2	Procedures	355
5.5.2.1	Modifying the hardware	355
5.5.2.1.1	Materials required	355

Table of contents

5.5.2.1.2	Customize the board	355
5.5.2.2	Modifying the software	359
5.5.2.2.1	ModusToolbox™ software	359
5.5.2.2.2	Serial memory interface (SMIF)	359
5.5.2.2.3	Import ModusToolbox™ QSPI flash code example	360
5.5.2.2.4	Determine the serial flash operation structure	363
5.5.2.2.5	Add additional flash operation commands	367
5.5.3	PSoC™ 6 MCU board operation	373
5.5.4	Use cases	375
5.5.5	Conclusion	377
5.5	References	378
5.5.6	Revision history	379
5.6	AN228740 Usage of Quad SPI (QSPI)/Serial Memory Interface (SMIF) in PSoC™ 6 MCU	379
5.6	About this document	379
5.6.1	Introduction	380
5.6.2	Getting started with QSPI	381
5.6.2.1	Using the Serial Flash Library	381
5.6.2.2	Using the Peripheral Driver Library	382
5.6.3	Features of QSPI	387
5.6.3.1	Clock domains	387
5.6.3.2	Modes	388
5.6.3.2.1	Command mode	388
5.6.3.2.2	XIP mode	388
5.6.3.3	Caches	388
5.6.3.4	Memory device signal interface	389
5.6.3.5	Cryptography	390
5.6.3.5.1	Cryptography in XIP mode	390
5.6.3.5.2	Cryptography in Command mode	392
5.6.4	Ecosystem	393
5.6.4.1	ModusToolbox™ Application Software libraries	394
5.6.4.2	QSPI Configurator tool	394
5.6.4.3	Programming tools	395
5.6.5	Configuration	396
5.6.5.1	QSPI configuration structure architecture	396
5.6.5.2	Configuration procedure	397
5.6.5.2.1	SFDP detection	397
5.6.5.2.2	Manual configuration	398
5.6.6	Order of operations	399
5.6.7	Programming external memory	400
5.6.8	Security with QSPI	402
5.6.9	Performance	403
5.6.10	Summary	404

Table of contents

5.6	Related documents	405
5.6.11	Revision history	406
5.7	AN230938 PSoC™ 6 MCU low-power analog	406
5.7	About this document	406
5.7.1	Introduction	407
5.7.2	Programmable analog features	408
5.7.2.1	Differences with other PSoC™ 6 devices	409
5.7.2.2	Analog routing	409
5.7.3	Low-power mode operation of analog resources	411
5.7.3.1	SAR ADCs	411
5.7.3.1.1	Clock source and limitations	413
5.7.3.1.2	Input options	413
5.7.3.1.3	Interrupts	413
5.7.3.2	Opamp	414
5.7.3.2.1	Power modes	415
5.7.3.3	DAC	415
5.7.3.4	Analog reference (AREF)	416
5.7.3.5	Low-power comparator (LPCOMP)	416
5.7.4	Power optimization techniques	417
5.7.4.1	Power modes of the block	417
5.7.4.2	Duty-cycling the blocks	417
5.7.4.2.1	Example: SAR ADC	417
5.7.4.2.2	Example: SAR ADC + opamp	418
5.7.4.3	SAR ADC scan frequency	420
5.7.4.4	SAR ADC acquisition time	421
5.7.4.5	SAR ADC reference buffer	422
5.7.4.6	LPOSC or medium frequency clock	423
5.7.4.7	LPOSC duty-cycling	423
5.7.4.8	Power-up delay	423
5.7.4.9	FIFO chain	423
5.7.4.10	DMA	423
5.7.5	Summary	424
5.7	References	425
5.7.6	Revision history	426
5.8	AN85951 PSoC™ 4 and PSoC™ 6 MCU CAPSENSE™ design guide	426
5.8	About this document	426
5.8.1	Introduction	427
5.8.1.1	Overview	427
5.8.1.2	CAPSENSE™ features	427
5.8.1.3	PSoC™ 4 and PSoC™ 6 MCU CAPSENSE™ Plus features	428
5.8.1.4	CAPSENSE™ design flow	429
5.8.2	CAPSENSE™ technology	432

Table of contents

5.8.2.1	CAPSENSE™ fundamentals	432
5.8.2.1.1	Self-capacitance sensing	433
5.8.2.1.2	Mutual-capacitance sensing	435
5.8.2.2	Capacitive touch sensing method	436
5.8.2.2.1	CAPSENSE™ sigma delta (CSD)	437
5.8.2.2.2	CAPSENSE™ crosspoint (CSX)	437
5.8.2.3	Signal-to-noise ratio (SNR)	439
5.8.2.4	CAPSENSE™ widgets	440
5.8.2.4.1	Buttons (zero-dimensional)	440
5.8.2.4.2	Sliders (one-dimensional)	444
5.8.2.4.3	Touchpads/Trackpads (two-dimensional)	445
5.8.2.4.4	Proximity (three-dimensional)	446
5.8.2.5	Liquid tolerance	447
5.8.2.5.1	Liquid tolerance for self capacitance sensing	448
5.8.2.5.2	Liquid tolerance for mutual-capacitance sensing	454
5.8.2.5.3	Effect of liquid properties on liquid-tolerance performance	456
5.8.3	PSoC™ 4 and PSoC™ 6 MCU CAPSENSE™	458
5.8.3.1	CAPSENSE™ generations in PSoC™ 4 and PSoC™ 6	458
5.8.3.2	CAPSENSE™ CSD sensing method (third and fourth generation)	460
5.8.3.2.1	GPIO cell capacitance to current converter	461
5.8.3.2.2	IDAC sourcing mode	462
5.8.3.2.3	IDAC sinking mode	464
5.8.3.2.4	CAPSENSE™ clock generator	465
5.8.3.2.5	Sigma-delta converter	466
5.8.3.2.6	Analog multiplexer (AMUX)	467
5.8.3.2.7	CAPSENSE™ CSD shielding	467
5.8.3.3	CAPSENSE™ CSX sensing method (third- and fourth- generation)	468
5.8.3.4	CAPSENSE™ CSD-RM sensing method (fifth-generation)	470
5.8.3.4.1	GPIO cell capacitance to charge converter	471
5.8.3.4.2	Capacitor DACs (CDACs)	473
5.8.3.4.3	CAPSENSE™ clock generator	473
5.8.3.4.4	Ratiometric sensing technology	474
5.8.3.4.5	Analog multiplexer (AMUX) and control matrix (CTRLMUX)	475
5.8.3.4.6	CAPSENSE™ CSDRM shielding	475
5.8.3.5	CAPSENSE™ CSX-RM sensing method (fifth-generation)	476
5.8.3.5.1	Ratiometric sensing technology	478
5.8.3.6	Autonomous scanning	479
5.8.3.7	Usage of multiple channels	479
5.8.4	CAPSENSE™ design and development tools	480
5.8.4.1	PSoC™ Creator	480
5.8.4.1.1	CAPSENSE™ component	480
5.8.4.1.2	CapSense_ADC component	481

Table of contents

5.8.4.1.3	Tuner GUI	481
5.8.4.1.4	Example projects	481
5.8.4.2	ModusToolbox™	482
5.8.4.2.1	CAPSENSE™ middleware	482
5.8.4.2.2	CAPSENSE™ configurator	482
5.8.4.2.3	CSDADC middleware	483
5.8.4.2.4	CSDIDAC middleware	483
5.8.4.2.5	CAPSENSE™ Tuner	483
5.8.4.2.6	Example projects	484
5.8.4.3	Hardware kits	484
5.8.5	CAPSENSE™ performance tuning	486
5.8.5.1	Selecting between SmartSense and manual tuning	486
5.8.5.2	SmartSense	486
5.8.5.2.1	Overview	486
5.8.5.2.2	SmartSense full auto-tune	488
5.8.5.2.3	SmartSense hardware parameters-only mode	492
5.8.5.2.4	SmartSense for initial tuning	492
5.8.5.3	Manual tuning	493
5.8.5.3.1	Overview	493
5.8.5.3.2	CSD sensing method (third- and fourth-generation)	494
5.8.5.3.3	CSX sensing method (third- and fourth-generation)	519
5.8.5.3.4	CSD-RM sensing method (fifth-generation)	527
5.8.5.3.5	CSX-RM sensing method (Fifth-generation)	551
5.8.5.3.6	Manual tuning trade-offs	558
5.8.5.3.7	Tuning debug FAQs	560
5.8.6	Gesture in CAPSENSE™	568
5.8.6.1	Touch gesture support	568
5.8.6.2	Gesture groups	568
5.8.6.3	One-finger gesture implementation	568
5.8.6.3.1	Tuning the widget	568
5.8.6.3.2	Selecting predefined gesture	569
5.8.6.3.3	Firmware implementation with timestamp	570
5.8.6.3.4	Tuning gesture parameters	570
5.8.6.3.5	Two-finger gesture implementation	576
5.8.6.3.6	Advanced filters for gestures	577
5.8.7	Design considerations	578
5.8.7.1	Firmware	578
5.8.7.1.1	Low-power design	580
5.8.7.2	Sensor construction	582
5.8.7.3	Overlay selection	583
5.8.7.3.1	Overlay material	583
5.8.7.3.2	Overlay thickness	583

Table of contents

5.8.7.3.3	Overlay adhesives	584
5.8.7.4	PCB layout guidelines	584
5.8.7.4.1	Sensor CP	584
5.8.7.4.2	Board layers	585
5.8.7.4.3	Button design	585
5.8.7.4.4	Slider design	593
5.8.7.4.5	Sensor and device placement	602
5.8.7.4.6	Trace length and width	602
5.8.7.4.7	Trace routing	602
5.8.7.4.8	Crosstalk solutions	604
5.8.7.4.9	Vias	605
5.8.7.4.10	Ground plane	606
5.8.7.4.11	Power supply layout recommendations	610
5.8.7.4.12	Layout guidelines for liquid tolerance	611
5.8.7.4.13	Schematic rule checklist	614
5.8.7.4.14	Layout rule checklist	618
5.8.7.5	Noise in CAPSENSE™ system	621
5.8.7.5.1	Finger injected noise	621
5.8.7.5.2	VDDA noise	623
5.8.7.5.3	External noise	624
5.8.7.6	Effect of grounding	640
5.8.7.6.1	CSX method	640
5.8.7.6.2	CSD method	643
5.8.8	CAPSENSE™ Plus	645
5.8.9	Resources	650
5.8.9.1	Website	650
5.8.9.2	Device datasheet	650
5.8.9.3	Component datasheet/middleware document	650
5.8.9.4	Technical reference manual	650
5.8.9.5	Development kits	650
5.8.9.6	PSoC™ Creator	650
5.8.9.7	ModusToolbox™	650
5.8.9.8	Application notes	651
5.8.9.9	Design support	651
5.8.10	Revision history	658
5.9	AN219528 PSoC™ 6 MCU low-power modes and power reduction techniques	662
5.9	About this document	662
5.9.1	Introduction	663
5.9.2	Power modes	664
5.9.2.1	Power mode transitions	664
5.9.2.2	CPU sleep and wakeup instructions	666
5.9.2.3	Low-power assistant	666

Table of contents

5.9.2.3.1	Low-power assistant features	666
5.9.2.3.2	PSoC™ 6 MCU device LPA settings	667
5.9.2.3.3	Connectivity device LPA software settings	667
5.9.2.4	Subsystem availability and power consumption	670
5.9.2.4.1	Subsystem availability	670
5.9.2.4.2	Approximating power consumption	670
5.9.2.4.3	Power estimator	670
5.9.2.5	Example case scenarios	671
5.9.2.6	System power management (SysPm) library	672
5.9.2.6.1	Overview	672
5.9.2.6.2	Mode transition functions	672
5.9.3	PSoC™ 6 MCU power management	675
5.9.3.1	Core voltage selection	675
5.9.3.1.1	Linear regulator and buck regulator	675
5.9.3.2	ULP mode clock	676
5.9.3.3	Backup power domain	677
5.9.3.3.1	V_{Backup} supply	677
5.9.3.3.2	Backup domain reset	681
5.9.3.3.3	External PMIC control	681
5.9.3.3.4	Wakeup sources	683
5.9.3.3.5	Backup data registers	684
5.9.4	Other power saving techniques	685
5.9.4.1	Use PSoC™ 6 MCU to gate current paths	685
5.9.4.2	Disable unused blocks	686
5.9.4.3	Use DMA to move data	686
5.9.4.4	Periodic wakeup timers	686
5.9.4.5	Disabling CPUs	687
5.9.4.6	Splitting tasks between the CPUs	687
5.9.4.7	Clocks	688
5.9.4.8	GPIOs	690
5.9.4.9	SRAM	691
5.9.4.10	TCPWM	691
5.9.4.11	SCB	692
5.9.4.12	Audio subsystem	693
5.9.4.13	USB	694
5.9.4.14	Low-power comparator	694
5.9.4.15	SAR ADC	694
5.9.4.16	Voltage DAC	695
5.9.4.17	Opamp	695
5.9.5	Power measurement	696
5.9.5.1	Measuring the current with a DMM	696
5.9.5.2	Approximating the power consumption	696

Table of contents

5.9.6	Power supply protection system	697
5.9.6.1	Hardware control	697
5.9.6.1.1	Brownout detect (BOD)	697
5.9.6.1.2	Low-voltage detect (LVD)	697
5.9.6.1.3	Overvoltage detect (OVD)	697
5.9.7	Summary	698
A	Power modes summary	699
A.1	Power modes and wakeup source	699
B	Subsystem availability	701
B.1	Resources available in different power modes	701
C	Callback function examples	703
C.1	Register callback functions	703
C.2	Implement custom callback functions	704
D	Code examples	705
D.1	CE219881 - PSoC™ 6 MCU switching between power modes	705
D.2	CE218129 - PSoC™ 6 MCU wakeup from hibernate using a low-power comparator	705
D.3	CE218542 - PSoC™ 6 MCU custom tick timer using RTC alarm interrupt	706
D.4	CE226306 - PSoC™ 6 MCU power measurements	707
5.9.12	References	710
5.9.13	Revision history	711
5.10	AN215656 PSoC™ 6 MCU dual-core system design	711
5.10	About this document	711
5.10.1	Introduction	712
5.10.1.1	Tools and libraries	714
5.10.1.2	How to use this document	715
5.10.2	General dual-core concepts	716
5.10.3	PSoC™ 6 MCU dual-core architecture	717
5.10.4	PSoC™ 6 MCU dual-core development	719
5.10.4.1	ModusToolbox™ software instructions	719
5.10.4.1.1	Creating a dual-core application	719
5.10.4.1.2	Customizing linker scripts	723
5.10.4.1.3	Sharing libraries and peripherals	723
5.10.4.2	PSoC™ Creator instructions	725
5.10.4.3	Resource assignment considerations	728
5.10.4.4	Power mode transition considerations	728
5.10.4.5	IPC configuration considerations	729
5.10.4.6	Interrupt assignment considerations	729
5.10.4.7	Debug considerations	730
5.10.4.7.1	ModusToolbox™ instructions	731
5.10.4.7.2	PSoC™ Creator instructions	731
5.10.4.7.3	Instructions for other IDEs	731
5.10.5	Summary	749

Table of contents

5.10	References	750
5.10.6	Revision history	752
5.11	AN215671 PSoC™ 6 MCU firmware design for BLE applications	752
5.11	About this document	752
5.11.1	Introduction	754
5.11.2	PSoC™ resources	755
5.11.3	BLE protocol implementation in PSoC™ 6 BLE	756
5.11.3.1	BLE host	757
5.11.3.1.1	Generic access profile (GAP)	757
5.11.3.1.2	Generic attribute profile (GATT)	758
5.11.3.1.3	ATT protocol – organizing the data	759
5.11.3.2	BLE controller	761
5.11.3.2.1	Managing multiple connections	762
5.11.4	Developing a BLE application: Firmware low	763
5.11.4.1	Implementing low-power BLE design	773
5.11.4.2	Implementing a secure BLE design	776
5.11.4.2.1	Configuring security features using the BLE component security mode and security level	777
5.11.4.2.2	Establishing secure BLE link: Firmware flow	779
5.11.4.3	Additional BLE design considerations	783
5.11.5	BLE design examples	785
5.11.5.1	Multi-master multi-slave: Implementing four BLE slaves	785
5.11.5.1.1	About the design	785
5.11.5.2	Multi-master multi-slave: Implementing three BLE masters and one BLE slave	787
5.11.5.2.1	About the design	788
5.11.6	Summary	790
	References	791
5.11.7	Revision history	792
5.12	AN217666 PSoC™ 6 MCU interrupts	792
5.12	About this document	792
5.12.1	Introduction	793
5.12.1.1	How to use this document	793
5.12.2	PSoC™ 6 MCU interrupt architecture	794
5.12.2.1	CY8C61x6/7, CY8C62x6/7, and CY8C63xx interrupt architecture	795
5.12.2.2	CY8C62x4, CY8C62x5, and CY8C62x8/A interrupt architecture	796
5.12.2.3	Types of interrupts	796
5.12.2.3.1	Level and pulse interrupts	797
5.12.2.4	Interrupts and Power modes	797
5.12.2.5	CPU sleep and wakeup	798
5.12.3	Interrupt configuration	800
5.12.3.1	Configuring interrupts using ModusToolbox™	800
5.12.3.1.1	Using HAL	800

Table of contents

5.12.3.1.2	Using Device Configurator and PDL	801
5.12.3.2	Configuring interrupts using PDL	801
5.12.3.3	Configuring interrupts using PSoC™ Creator	804
5.12.3.3.1	Using the schematic (TopDesign)	804
5.12.3.3.2	Using the design-wide resource window (CyDWR)	806
5.12.3.3.3	Using PSoC™ Creator generated code and PDL	807
5.12.4	Debugging tips	809
5.12.5	Advanced interrupt topics	810
5.12.5.1	Exceptions	810
5.12.5.2	Interrupt latency	811
5.12.5.3	Nested interrupts	812
5.12.5.4	Code optimization	812
5.12	References	814
A	Appendix A. Interrupt sources in PSoC™ 6 MCU	816
5.12.7	Revision history	819
5.13	AN218241 PSoC™ 6 MCU hardware design considerations	819
5.13	About this document	819
5.13.1	Introduction	820
5.13.2	Package selection	821
5.13.3	Power	822
5.13.3.1	Buck regulator inductor/capacitor selection	822
5.13.3.2	Power pin connections	823
5.13.3.3	PMIC controller	826
5.13.3.4	Power ramp-up and sequencing considerations	826
5.13.3.5	Device power settings	826
5.13.3.6	Thermal considerations	828
5.13.3.7	eFuse programming	828
5.13.4	Clocking	829
5.13.4.1	Clock settings	829
5.13.4.1.1	Crystal oscillators	830
5.13.4.1.2	External clock	833
5.13.5	Reset	836
5.13.6	Programming and debugging	837
5.13.6.1	SWD	837
5.13.6.2	JTAG	837
5.13.6.3	ETM	838
5.13.6.4	Debug select	838
5.13.7	GPIO pins	840
5.13.7.1	I/O pin selection	840
5.13.8	Analog module design tips	844
5.13.8.1	CAPSENSE™	844
5.13.8.2	SAR ADC	847

Table of contents

5.13.8.2.1	SAR ADC acquisition time	848
5.13.8.3	CTDAC	849
5.13.9	Using external memory in the design	850
5.13.10	USB connection	851
5.13.10.1	PSoC™ 6 MCU USB pin description	851
5.13.10.2	PSOC™ 6 MCU as USB device	851
5.13.11	Antenna design	852
5.13.11.1	Support for external power amplifier/low-noise amplifier/RF front-end	854
5.13.12	Audio subsystem	856
5.13.12.1	Clock generation for PDM-PCM converter	856
5.13.12.2	Clock generation for I2S audio devices	857
5.13.13	Secure digital host controller	858
5.13.14	Summary	860
	References	861
A	PCB layout tips	862
B	Schematic checklist	863
5.13.17	Revision history	865
5.14	AN219434 PSoC™ 6 MCU importing generated code into an IDE	865
5.14	About this document	865
5.14.1	Introduction	867
5.14.2	Integrating generated code	868
5.14.2.1	Advantages of generated code	870
5.14.2.2	Limitations of generated code	870
5.14.2.3	Tool compatibility	871
5.14.3	Exporting and importing generated code	872
5.14.3.1	Enabling export for a target IDE	873
5.14.3.2	Generating the export package	873
5.14.3.3	Importing the package	873
5.14.3.4	Supporting iterative development	874
5.14.3.5	Export/Import review	875
5.14.4	Manually importing generated code	876
5.14.4.1	Creating a project file	876
5.14.4.2	Locating and identifying source files	876
5.14.4.2.1	User files	877
5.14.4.2.2	Design files	879
5.14.4.2.3	Library files	880
5.14.4.3	Adding files to a project	881
5.14.4.3.1	Where to get PDL user files	881
5.14.4.3.2	Where to get PDL library files	882
5.14.4.4	Supporting iterative development	882
5.14.4.5	Manual import review	882
5.14.5	Manually importing generated code – an example	883

Table of contents

5.14.5.1	Complete the design and generate application files	883
5.14.5.2	Set up the IDE project file	884
5.14.5.3	Add files to the project	886
5.14.5.3.1	Add firmware files	887
5.14.5.3.2	Add user files	887
5.14.5.3.3	Add design files	888
5.14.5.3.4	Add PDL library files	889
5.14.5.3.5	Add Bluetooth® Low Energy library files	891
5.14.5.4	Set include paths	893
5.14.5.4.1	Set a path to user files	893
5.14.5.4.2	Set a path to design files	894
5.14.5.4.3	Set a path to peripheral folder (driver files)	895
5.14.5.4.4	Set a path to the middleware folder (Bluetooth® Low Energy stack)	896
5.14.5.4.5	Set other required paths	897
5.14.5.5	Build the project in the IDE	897
5.14.5.6	Example review	898
5.14.6	Summary	899
5.14	References	900
A	Appendix A - configuring an IDE project file	901
B	Appendix B - using generated code for learning	902
5.14.9	Revision history	903
5.15	AN221111 PSoC™ 6 MCU designing a custom secured system	903
5.15	About this document	903
5.15.1	Introduction	904
5.15.2	System security	905
5.15.2.1	Security basics	907
5.15.2.1.1	External access	907
5.15.2.1.2	Internal access	907
5.15.2.2	Basic definitions	908
5.15.3	Security features	911
5.15.3.1	eFuse	911
5.15.3.2	Device lifecycle	913
5.15.3.2.1	VIRGIN	913
5.15.3.2.2	NORMAL	913
5.15.3.2.3	SECURE	915
5.15.3.2.4	SECURE WITH DEBUG	915
5.15.3.2.5	RMA	916
5.15.3.3	Protection state	916
5.15.3.4	CM0+ boot sequence	916
5.15.3.4.1	NORMAL lifecycle boot	919
5.15.3.4.2	SECURE lifecycle boot	919
5.15.3.4.3	SECURE_WITH_DEBUG lifecycle stage	920

Table of contents

5.15.3.4.4	NORMAL lifecycle with validate	920
5.15.3.4.5	Debug boot errors	920
5.15.3.5	Chain of Trust (CoT)	920
5.15.3.6	Code signing and verification	922
5.15.3.6.1	Code signing	922
5.15.3.6.2	Code verification	923
5.15.3.7	Protection units	924
5.15.3.7.1	SMPUs	925
5.15.3.7.2	PPUs	926
5.15.3.7.3	Protection unit configuration	926
5.15.3.7.4	Bus masters	927
5.15.3.7.5	Protection contexts (PC)	927
5.15.3.7.6	SMPU/PPU master	928
5.15.3.8	Debug port configuration	929
5.15.3.8.1	Debug port architecture	929
5.15.3.8.2	System access port (SYS-AP)	931
5.15.4	Project configuration	932
5.15.4.1	Table of Contents2 (TOC2)	933
5.15.4.2	CYPRESS™ standard application format	938
5.15.4.3	Infineon secured boot RSA public key format	940
5.15.4.4	Programming eFuse to change the lifecycle stage	942
5.15.4.4.1	Using CYPRESS™ Programmer	943
5.15.4.4.2	Setting up CYPRESS™ Programmer	943
5.15.5	Dual CPU design considerations	944
5.15.6	Summary	945
5.15.7	Appendix A - Code example of a security application template	946
5.15.7.1	Bootup flow	947
5.15.7.2	Application Chain of Trust (CoT)	949
5.15.7.3	Project memory map	950
5.15.8	Appendix B - Creating crypto key pairs	952
5.15.8.1	Generating the RSA key pair	952
5.15.8.2	Generating the ECC key pair	952
5.15.8.2.1	Example RSA private/public key files	953
5.15.8.3	Editing the RSA key C file (cy_ps_keystorage.c)	954
5.15.9	Appendix C - Debug port access settings	959
5.15.9.1	Debug access settings	959
5.15.9.2	Firmware control of Debug Port	960
5.15.9.2.1	1 st generation PSoC™ 6 devices	961
5.15.9.2.2	2 nd generation PSoC™ 6 devices	961
5.15.9.2.3	Configure SWJ for Debug	962
5.15.9.3	eFuse programming for debug access restrictions and lifecycle stage	962
5.15.10	Appendix D - Transition to RMA	966

Table of contents

5.15.11	Appendix E - Protection unit configuration	967
5.15.11.1	Example Protection unit configuration	967
5.15.11.2	Pre-configured protection units	969
5.15.12	Appendix F - Debug codes for failed boot sequences	970
5.15	Related documents	971
5.15.13	Revision history	972
5.16	AN213924 PSoC™ 6 MCU Device Firmware Update (DFU) software development kit guide	972
5.16	About this document	972
5.16.1	Introduction	973
5.16.2	What Is device firmware update?	974
5.16.2.1	Terms and definitions	974
5.16.2.2	Using a DFU module	975
5.16.2.3	Basic DFU function flow	975
5.16.2.4	Other use cases	976
5.16.3	DFU SDK description	978
5.16.3.1	DFU SDK files	978
5.16.3.1.1	User callback functions	979
5.16.3.1.2	Linker scripts	980
5.16.3.2	DFU ecosystem	980
5.16.4	How to use the SDK	982
5.16.4.1	Determine the applications in your system	982
5.16.4.2	Locate applications in memory	983
5.16.4.3	Design the applications	984
5.16.4.3.1	ModusToolbox™ instructions	984
5.16.4.3.2	PSoC™ Creator instructions	987
5.16.4.4	Build and program the applications	992
5.16.5	PSoC™ 6 MCU DFU code examples	994
5.16.5.1	How to build the code examples	995
5.16.5.2	PSoC™ 6 MCU basic DFUs	995
5.16.5.2.1	ModusToolbox™ instructions	995
5.16.5.2.2	PSoC Creator instructions	998
5.16.5.3	PSoC™ 6 MCU BLE bootloaders	1005
5.16.5.3.1	BLE bootloader with upgradeable stack	1010
5.16.5.4	PSoC™ 6 MCU dual-application bootloader	1012
5.16.5.5	PSoC™ 6 MCU encrypted bootloader	1014
5.16	References	1018
A	DFU host tool	1020
B	Host command/response protocol	1021
B.1	Command/Response packet structure	1021
B.2	Commands	1021
B.2.1	Enter DFU	1022
B.2.2	Sync DFU	1022

Table of contents

B.2.3	Exit DFU	1023
B.2.4	Send data	1023
B.2.5	Send data without response	1023
B.2.6	Program data	1023
B.2.7	Verify data	1024
B.2.8	Erase data	1024
B.2.9	Verify application	1025
B.2.10	Set application metadata	1025
B.2.11	Get metadata	1026
B.2.12	Set ElVector	1026
C	.cyacd2 file format	1028
D	Application metadata	1029
E	Metadata structure	1031
F	Post-build file listings	1032
F.1	ModusToolbox™ post-build bash file	1032
F.2	PSoC™ Creator post-build batch file	1033
5.16.12	Revision history	1034
5.17	AN235691 ModusToolbox™ & Friends	1034
5.17	About this document	1034
5.17.1	Introduction	1035
5.17.1.1	What is a partner?	1035
5.17.1.2	Why become a partner?	1035
5.17.1.3	How to become a partner?	1035
5.17.2	ModusToolbox™ software overview	1036
5.17.2.1	Types of software content	1036
5.17.2.1.1	Code examples	1036
5.17.2.1.2	Middleware	1038
5.17.2.1.3	Board support packages (BSPs)	1041
5.17.2.2	Manifests	1041
5.17.2.3	Git versioning control system	1043
5.17.2.4	How does everything come together in ModusToolbox™?	1043
5.17.2.5	How does partner integration work?	1044
5.17.3	Setting up your own Git infrastructure	1045
5.17.3.1	Choosing your Git hosting platform	1045
5.17.3.2	Choosing your development flow	1045
5.17.3.2.1	Single-stage workflow	1045
5.17.3.2.2	Dual-stage workflow	1046
5.17.3.3	Designing your internal staging setup	1047
5.17.3.4	Designing your external production setup	1047
5.17.4	Creating your own software content	1048
5.17.4.1	Creating a code example	1048
5.17.4.1.1	Choosing a starter application	1048

Table of contents

5.17.4.1.2	Choosing the name	1048
5.17.4.1.3	Choosing the title	1050
5.17.4.1.4	Adding the source files	1050
5.17.4.1.5	Adding middleware	1050
5.17.4.1.6	Adding the End User License Agreement (EULA)	1051
5.17.4.1.7	Create a Git repository	1051
5.17.4.1.8	Create a topic branch	1051
5.17.4.1.9	Testing the code example	1051
5.17.4.1.10	Merging into mainline	1052
5.17.4.1.11	Creating the release package	1052
5.17.4.2	Creating a middleware library	1053
5.17.4.3	Creating a BSP	1053
5.17.5	Creating your own manifest	1054
5.17.5.1	Creating repositories	1054
5.17.5.2	Creating your code example manifest	1055
5.17.5.2.1	Adding BSP capabilities	1060
5.17.5.2.2	Specifying requirements for restricted scope	1063
5.17.5.3	Creating your middleware manifest	1066
5.17.5.3.1	Adding middleware dependencies	1070
5.17.5.4	Creating your BSP manifest	1073
5.17.5.4.1	Adding BSP dependencies	1078
5.17.5.5	Creating your super manifest	1082
5.17.5.6	Specifying the dependency manifests	1085
5.17.5.7	Validating your manifests	1085
5.17.6	Testing the manifest integration	1086
5.17.6.1	Testing the dependency manifest integration	1088
5.17.6.2	Out-of-the-box testing	1090
5.17.7	Integrating into ModusToolbox™	1091
5.17.8	Updating your content	1092
5.17.8.1	Clone repositories	1092
5.17.8.2	Create a topic branch	1092
5.17.8.3	Update and test your content	1092
5.17.8.4	Merge into mainline	1092
5.17.8.5	Create release package	1092
5.17.8.6	Update corresponding manifest	1093
5.17.8.7	Update dependency manifest	1094
5.17.8.8	Testing the integration	1094
5.17.9	Technical support	1095
5.17.10	Summary	1098
5.17.11	Appendix A – Partners integrated into ModusToolbox™	1099
5.17.11.1	Memfault	1099
5.17.12	Revision history	1100

Table of contents

5.18	AN235279 Performing ETM and ITM trace on PSoC™ 6 MCU	1100
5.18	About this document	1100
5.18.1	Introduction	1101
5.18.1.1	What is trace?	1101
5.18.1.2	Why is trace important?	1101
5.18.1.3	Overview of subsequent chapters	1101
5.18.2	General and PSoC™ 6 MCU Arm® trace architecture	1102
5.18.2.1	General trace architecture	1102
5.18.2.1.1	Trace Source	1102
5.18.2.1.2	Trace Sink	1103
5.18.2.1.3	Trace Link	1103
5.18.2.2	Trace output	1104
5.18.2.2.1	On-chip capture	1104
5.18.2.2.2	Off-chip capture	1104
5.18.2.3	Trace infrastructure examples	1105
5.18.2.3.1	Single-core, off-chip ETM trace example	1105
5.18.2.3.2	Multi-core, off-chip ETM trace example	1105
5.18.2.3.3	Multi-core, off-chip ETM CTI trace example	1105
5.18.2.4	PSoC™ 6 MCU trace infrastructure	1106
5.18.3	Hardware and software requirements	1107
5.18.3.1	Hardware requirements	1107
5.18.3.1.1	Trace probes (external debugger)	1107
5.18.3.1.2	Development board	1107
5.18.3.2	Software requirements	1108
5.18.4	Performing trace on PSoC™ 6 MCU	1109
5.18.4.1	Creating/importing project using ModusToolbox™	1109
5.18.4.2	Performing trace on IAR Embedded Workbench	1110
5.18.4.2.1	Import the ModusToolbox™ project into IAR EW	1110
5.18.4.2.2	Configure the debugger script and debugger	1111
5.18.4.2.3	Perform ETM trace	1112
5.18.4.2.4	Perform ITM trace (printf-style debugging)	1114
5.18.4.3	Performing trace on Keil µVision	1116
5.18.4.3.1	Import the ModusToolbox™ project into Keil µVision	1116
5.18.4.3.2	Configure the debugger script and debugger	1116
5.18.4.3.3	Perform ETM trace	1119
5.18.4.3.4	Perform ITM trace (printf-style debugging)	1120
5.18.5	Summary	1121
5.18	References	1122
5.18.6	Revision history	1123
5.19	AN235297 Creating a ModusToolbox™ 3.x BSP	1123
5.19	About this document	1123
5.19.1	Introduction	1124

Table of contents

5.19.1.1	What is a BSP?	1124
5.19.1.2	BSP Assistant overview	1124
5.19.1.3	Overview	1125
5.19.1.4	Software requirement	1125
5.19.2	BSP design	1126
5.19.2.1	Software	1126
5.19.2.1.1	Peripheral Driver Library (PDL)	1126
5.19.2.1.2	Hardware Abstraction Layer (HAL)	1126
5.19.2.1.3	Other libraries	1126
5.19.2.2	Documentation	1126
5.19.2.3	Typical BSP contents	1127
5.19.2.4	Startup code and linker files	1127
5.19.2.5	Configuration files	1128
5.19.2.6	Generated source files	1129
5.19.2.7	Static source files	1130
5.19.2.8	Documentation files	1130
5.19.3	Using the BSP Assistant tool	1131
5.19.3.1	Creating a new BSP	1131
5.19.3.1.1	Create and configure the BSP	1131
5.19.3.1.2	Create an application	1137
5.19.3.1.3	Code Build	1142
5.19.3.2	Customizing an existing BSP	1142
5.19.3.2.1	Create an application	1142
5.19.3.2.2	Customize the BSP	1144
5.19.3.2.3	Update dependencies and build	1146
5.19.4	Advanced usage	1148
5.19.4.1	Differences between ModusToolbox™ BSP generations	1148
5.19.4.2	Migrating the ModusToolbox™ BSP	1149
5.19.4.2.1	Using the BSP Assistant tool	1149
5.19.4.2.2	Without using the BSP Assistant tool	1150
5.19	References	1152
5.19.5	Revision history	1153
6	PSoC™ 6 development kits	1153
7	Other PSoC™ 6 documents	1154
7.1	PSoC™ CAD, BSDL, and IBIS files	1154
7.2	PSoC™ 6 technical reference manuals	1155
	Disclaimer	1157

~~1 PSoC™ 6 MCU overview~~

~~DRAFT~~ PSoC™ 6 MCU overview

The PSoC™ 6 microcontroller (MCU) family is a range of general-purpose MCUs built on an ultra-low-power architecture ideal for battery-operated, low-power applications including embedded IoT applications.

Key features

- **Dual-core CPU architecture based on ARM® Cortex®-M4 and Cortex®-M0+:** Lets designers optimize for power and performance simultaneously and use dedicated DMA controllers for data transfer operations
- **Security built into the platform:** Incorporates hardware-based Root of Trust (RoT), hardware cryptographic accelerators, isolated processing environments, secured onboarding, boot, key storage, and firmware updates, and trusted FW-M security services
- **Wide range of memory density options:** Supports on-chip flash memory from 256 KB to 2048 KB and on-chip SRAM from 128 KB to 1024 KB. Memory expansion capability is provided through the Quad SPI (QSPI) interface
- **Rich, low-power analog and digital peripherals:** Capacitive touch interface featuring 4th-generation CAPSENSE™, low-power analog such as 12-bit SAR ADCs, opamps, comparators, and DACs; digital peripherals such as PWM, serial protocols, USB 2.0 Full Speed, I2S, PDM-PCM, CAN-FD.
- **Comprehensive package options:** Supported packages include WLCSP, BGA, QFN, and TQFP. Up to 102 GPIOs are available for the application.

1 PSoC™ 6 MCU overview

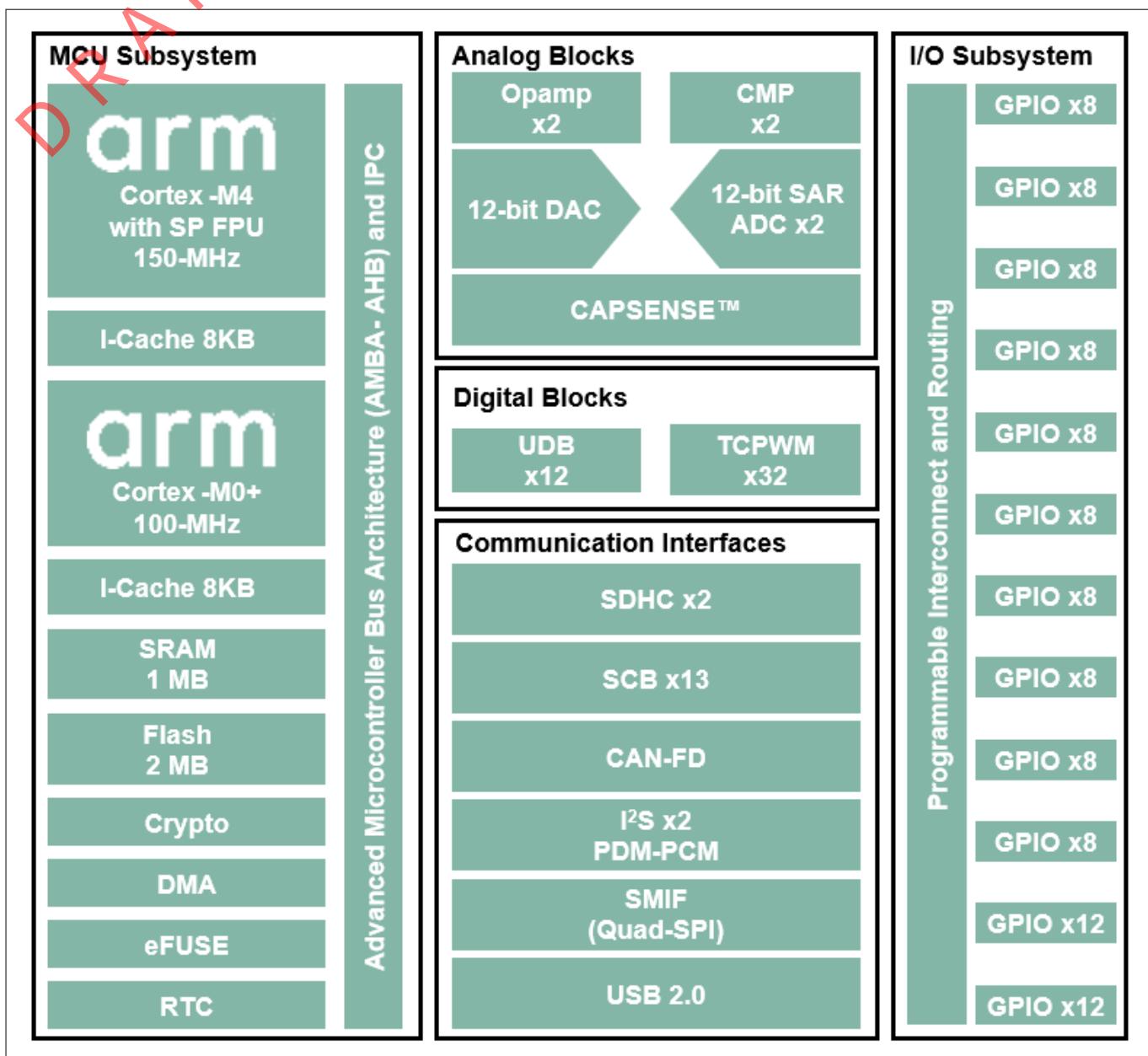


Figure 1 PSoC™ 6 MCU subsystems, blocks, and interfaces

PSoC™ 6 MCU family sub-categories

The PSoC™ 6 MCU product family is categorized into four sub-families:

- **PSoC™ 61 programmable line, which offers the PSoC™ 6 MCU in a single-CPU configuration** where only the ARM® Cortex® -M4 CPU is available for the user application
- **PSoC™ 62 performance line, which offers the PSoC™ 6 MCU in a dual-CPU configuration** where both the ARM® Cortex® -M4 and ARM® Cortex® -M0+ CPUs are available for the user application
- **PSoC™ 63 Bluetooth® Low Energy connectivity line**, which features the PSoC™ 6 MCUs with an integrated Bluetooth® LE radio
- **PSoC™ 64 secured MCU line** that incorporates all of the key features of PSoC™ 6 with preconfigured software to support secure onboarding, boot, firmware updates, and trusted FW-M security services

Target applications

- Wearables

~~DRAFT~~ 1 PSoC™ 6 MCU overview

- Smart home, home automation, home appliances
- Portable medical devices
- Battery-powered IoT products

Use cases

- **As the main application MCU:** PSoC™ 6 MCU interfaces with analog and digital sensors, drives user interfaces like capacitive touch interface and serial interface displays, and provides the application functionality such as motor control, sensor data processing, and edge-based machine learning.
- **As the main application MCU with Wi-Fi, Bluetooth® Low Energy connectivity:** Along with being the main application MCU, PSoC™ 6 also acts as the wireless host MCU hosting the Wi-Fi and Bluetooth® software stacks, handling OTA updates, and running the embedded IoT application that interfaces to the cloud.
- **As a coprocessor:** PSoC™ 6 MCU acts as a coprocessor to an application processor such as a Linux-based MPU. The MPU offloads tasks that require low-latency real-time processing, low-power always-on sensing, etc. to the PSoC™ 6 MCU. Examples of these tasks include analog/digital sensors data aggregation and transfer, hosting of wireless connectivity stacks, and cloud connectivity management.

ModusToolbox™ software, evaluation kits, documentation

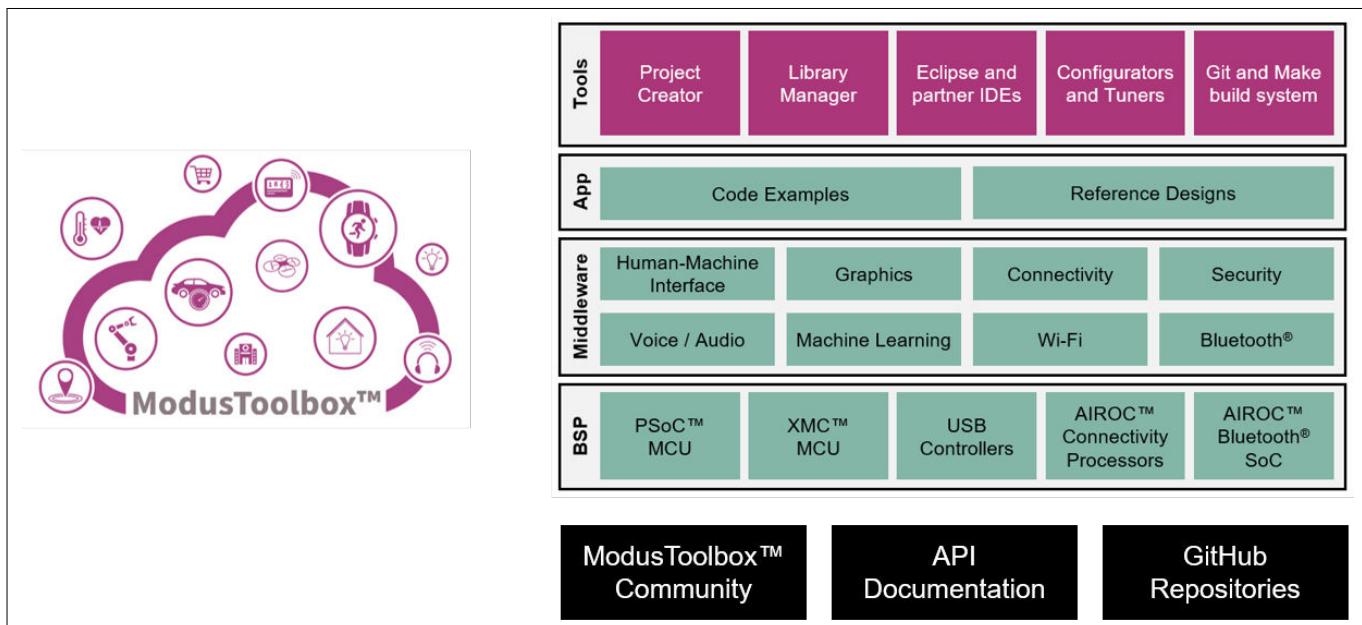


Figure 2 ModusToolbox™ ecosystem

PSoC™ 6 MCU application development is supported in the ModusToolbox™ software. ModusToolbox™ is a collection of easy-to-use software and tools enabling rapid development with Infineon MCUs, covering applications from embedded sense and control to wireless and cloud-connected systems using AIROC™ Wi-Fi, Bluetooth®, and combo devices. Hundreds of code examples are available to accelerate your application development, and reduce development time.

Infineon provides a wide variety of hardware platforms to enable developers to get started and prototype with PSoC™ 6 MCUs in ModusToolbox™ software. The hardware platforms include the low-cost, small-form-factor prototyping kits, and full-featured pioneer/evaluation kits that support hardware expansion through popular interfaces, such as Arduino Uno and mikroBUS.

Infineon provides quality technical documentation that enables developers to use PSoC™ 6 MCUs in the end products from the prototyping to the production. These include device datasheets, technical reference manuals, application notes covering hardware, software, solutions, and SDK documentation.

~~DO NOT USE~~ 2 PSoC™ 6 datasheets

2 PSoC™ 6 datasheets

Device datasheets provide a technical overview of the device that includes the key features, hardware architecture, on-chip peripherals, various sub-systems, and package details. Electrical specifications of the device are also provided in the datasheet.

Refer to the respective Technical Reference Manual (TRM) for detailed technical information on a device family. The datasheets for the PSoC™ 6 device families are listed below.

Device Family	Datasheet
PSOC™ 61 Programmable Line (Single Core Applications CPU: ARM® Cortex® -M4)	
CY8C61x4	PDF
CY8C61x5	PDF
CY8C61x6, CY8C61x7	PDF
CY8C61x8, CY8C61xA	PDF
PSOC™ 62 Performance Line (Dual Core Applications CPU: ARM® Cortex® -M4, ARM® Cortex® -M0+)	
CY8C62x4	PDF
CY8C62x5	PDF
CY8C62x6, CY8C62x7	PDF
CY8C62x8, CY8C62xA	PDF
PSOC™ 63 Bluetooth® Low Energy Connectivity Line (MCUs with on-chip Bluetooth® radio)	
CY8C63x6, CY8C63x7	PDF
PSOC™ 64 Secure MCU Line (Applications CPU: ARM® Cortex® -M4, Secure CPU: ARM® Cortex® -M0+)	
CYB064x5 Secure Boot	PDF
CYB064x7 Secure Boot	PDF
CYB064xA Secure Boot	PDF
CYB064x7-BL Secure Boot BLE	PDF
CYS064xA Standard Secure	PDF

Related information

[PSoC 6 technical reference manuals on page 1155](#)

~~3 PSoC™ 6 software tools~~

~~3 PSoC™ 6 software tools~~

ModusToolbox™ software is a modern, extensible development environment supporting a wide range of Infineon microcontroller devices including PSoC 6. This section includes all the documentation relevant to ModusToolbox™.

3.1 ModusToolbox™ installation guide

About this document

- 1

Scope and purpose

This guide provides instructions for installing the ModusToolbox™ tools package, version 3.0.0. This is a set of tools that enable you to integrate our devices into your existing development methodology. Refer to the [release notes](#) for details about what is included. Refer to earlier revisions of this guide for instructions to install previous versions of ModusToolbox™ tools packages.

Intended audience

This document helps application developers understand how to install the ModusToolbox™ tools package.

Reference documents

Refer to the [ModusToolbox™ user guide](#) for a description of the software and instructions to get started. You can also refer to the [ModusToolbox™ training available on GitHub](#).

If you plan to use the Eclipse IDE included with the ModusToolbox™ software, refer to these documents, which are also available from the Eclipse IDE Help menu:

- [Eclipse IDE for ModusToolbox™ quick start guide](#): brief instructions to create, build, and program applications.
- [Eclipse IDE for ModusToolbox™ user guide](#): more detailed information about using the IDE.

~~3 PSoC™ 6 software tools~~

~~3.1.1 General information~~

~~3.1.1.1 System requirements~~

The ModusToolbox™ software consumes approximately 2 GB of disk space. Like most modern software, it requires both free disk space and memory to run effectively. We recommend a system configuration with a PassMark CPU score > 2000 (cpubenchmark.net), at least 25 GB of free disk space, and 8 GB of RAM. The product will operate with fewer resources; however, performance may be degraded.

ModusToolbox™ software is supported on the following 64-bit operating systems:

Host OS	Supported	Recommended (full testing)
Windows	7 SP1, 10, 11	10
macOS	Catalina, Big Sur, Monterey (Intel processors)	Big Sur
	Big Sur, Monterey (Arm processors)	
Linux	Ubuntu 18.04 LTS, 20.04 LTS	20.04 LTS

Note: All the above operating systems are supported by all 3.x releases.

On macOS, Arm processors are supported via Rosetta. Support for Intel processors is guaranteed in all 3.x releases.

*ModusToolbox™ software is **not** supported on 32-bit operating systems.*

3.1.1.2 Uninstall Beta versions

If you installed any Beta release of ModusToolbox™ 3.0 software, you need to uninstall it before installing this release. To uninstall any Beta release:

- Windows: The current release installer will prompt you to uninstall a previous version 3.0 installation. You can also use the Windows Control Panel.
- Linux: Go to the directory where you extracted the tar.gz installer. Delete the docs_3.0, tools_3.0, and ide_3.0 directories, as well as EULA 3.0 text file from the "ModusToolbox" directory.
- macOS: The current release installer contains a check box to uninstall a previous version 3.0 installation.

Note: If you plan to use the Eclipse IDE for ModusToolbox™ included with the installation, be aware that uninstalling the ModusToolbox™ software does not remove any Eclipse IDE workspaces you may have previously created. You should manually delete these workspaces or move them to another location. See also [Step 3: Run the Eclipse IDE](#) for more details about workspaces.

~~3 PSoC™ 6 software tools~~

~~3.1.2 Step 1: Download the software~~

ModusToolbox™ software is available from the Infineon Developer Center website (<https://softwaretools.infineon.com/tools/com.ifx.tb.tool.modustoolbox>) to download and/or install.

Select the appropriate package for your operating system:

- Windows: ModusToolbox_3.0.0.<build>-windows-install.exe
- Linux: ModusToolbox_3.0.0.<build>-linux-install.tar.gz
- macOS: ModusToolbox_3.0.0.<build>-macos-install.pkg

You can then click **Download** or **Install**:

- If you click **Download**, the selected package will be downloaded to your computer. Refer to [Step 2](#) to install the software.
- If you have not used the website before, and if you click **Install**, a message will display asking you to install the Developer Center. Follow the instructions on the website to install the software.

3.1.2.1 ModusToolbox™ patches

Ensure you are downloading the core version "2.4.0" of the ModusToolbox™ tools package. There may be patch versions also available, but they will not work without the appropriate core version first installed. Also, patch version "2.3.1" will not work with core version "2.4.0," for example, because it is a patch for core version "2.3.0."

3.1.2.2 Prerequisites

ModusToolbox™ software requires the following Unix programs (with minimum versions) to work properly. On Windows, these are provided by the installer program. For macOS and Linux, you must install these programs as appropriate:

- cmp (v2.8.1)
- git (2.17.0)
- make (v3.81)
- mktemp (v8.25)
- perl (v5.18.2)
- python (v3.7)
- cysecuretools (v3.10) – Refer to the "[Secure Boot](#)" [SDK User Guide](#) for more details.

Some versions of Ubuntu Linux do not include 'make' by default. Use the following command to install it:

```
sudo apt-get install make
```

3.1.2.3 SEGGER J-Link

If you plan to use the SEGGER J-link debugger, you must download and install the appropriate software pack for your OS. It is not included with the ModusToolbox™ software. Use version 6.98 or later. For Linux, if you install this using the tar.gz file, make sure you install J-Link in a common location. Otherwise, you must configure the Eclipse IDE to specify the location, as follows:

Window > Preferences > MCU > Global SEGGER J-Link Path

- **Executable:** JLinkGDBServerCLExe
- **Folder:** <J-Link_extracted_location>

~~3 PSoC™ 6 software tools~~

~~3.1.3 Step 2: Install ModusToolbox™ software~~

~~Note:~~ Do not use spaces in the installation directory name. Various tools, such as Make, do not support spaces. Also, do not use common illegal characters, such as: / : * ? " < > |

~~Note:~~ If your user home directory contains spaces, see [Installing with spaces in user home directory](#).

3.1.3.1 Installing in non-default location

If you install ModusToolbox™ software in a non-default location, you will need to set the environment variable CY_TOOLS_PATHS to point to the <install_path>/ModusToolbox/tools_3.0 directory, or set that variable in each Makefile. You must use forward slashes in the variable's path, even in Windows. Refer to the "Product versioning" section in the [ModusToolbox™ user guide](#).

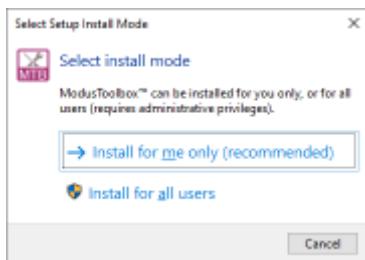
3.1.3.2 Installing with previous versions

ModusToolbox™ version 3.0 installs alongside previous versions of the software (version 2.4, 2.3, etc.); therefore, all versions can be used independently. However, be aware that various programs including the Eclipse IDE and the build system will detect and use the most current version of the "tools" directory by default. For example, if you have both versions 3.0 and 2.4 installed, and if you launch the Project Creator from the Eclipse IDE for version 2.4, it will open the version from the "tools_3.0" directory instead of the "tools_2.4" directory.

To control this behavior, use the environment variable CY_TOOLS_PATHS as described in the "Product Versioning" section in the [ModusToolbox™ user guide](#). This variable applies to all versions of ModusToolbox™ software, so you will have to update it as you work with different versions.

3.1.3.3 Windows

Run the ModusToolbox_3.0.0.<build>-windows-install.exe installer program and follow the prompts to install for the current user only or for all users of the same machine. For more information, see the [Default versus advanced Windows installation](#) section later in this document.



By default, ModusToolbox™ software is installed here:

C:\Users\<user_name>\ModusToolbox

~~Note:~~ If you have not installed ModusToolbox™ software previously, you may be prompted to restart your computer due to installation of Microsoft Visual C++ redistributable files.

~~3 PSoC™ 6 software tools~~

~~3.1.3.4 Linux~~

Extract the ModusToolbox_3.0.0.<build>-linux-install.tar.gz file to your <user_home> directory. The extraction process will create a "ModusToolbox" directory there, if there is not one there already.

After extracting, you must run the following scripts before running ModusToolbox™ software on your machine:

- OpenOCD: <user_home>/ModusToolbox/tools_3.0/openocd/udev_rules/install_rules.sh
- AIROC™ Bluetooth® Boards: <user_home>/ModusToolbox/tools_3.0/driver_media/install_rules.sh
- Firmware Loader: <user_home>/ModusToolbox/tools_3.0/fw-loader/udev_rules/install_rules.sh
- Post-Install Script: <user_home>/ModusToolbox/tools_3.0/modus-shell/postinstall
- IDC Registration Script: <user_home>/ModusToolbox/tools_3.0/idc_registration-2.4.0.bash

On Ubuntu systems, you must install additional packages using the following command:

```
$ sudo apt install libncurses5 libusb-1.0-0 libxcb-xinerama0
```

3.1.3.5 macOS

Double-click the downloaded ModusToolbox_3.0.0.<build>-osx-install.pkg file and follow the wizard.

The ModusToolbox™ software will be installed under the **Applications** folder in the volume you select in the wizard.

Note: *The ModusToolbox™ package installer installs a custom USB driver for use with ModusToolbox™ on macOS versions prior to Catalina. It may pop up a "System Extension Blocked" dialog. In this case, go to **Security Preference** and click **Allow** for the driver to be installed.*

In order for ModusToolbox™ to work correctly on macOS, you must install an additional Xcode package if you do not already have it installed. We recommend you install Xcode using the following command in a terminal window:

```
xcode-select --install
```

Note: *You may install Xcode from the App Store, but it will likely consume much more disk space than using the above command.*

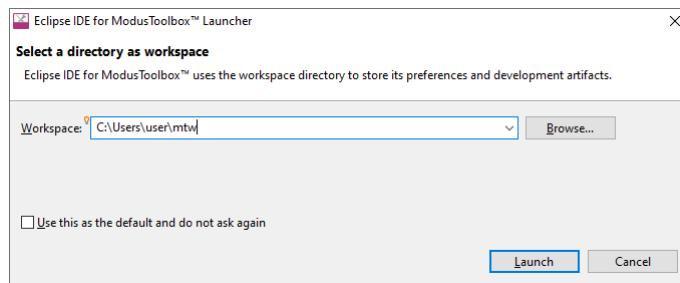
~~3 PSoC™ 6 software tools~~

~~3.1.4 Step 3: Run the Eclipse IDE~~

The ModusToolbox™ software includes an optional Eclipse IDE. To run the IDE:

- Windows: The installer provides an option to run the Eclipse IDE on the final step. You can also select the Eclipse IDE for ModusToolbox™ 3.0 item from the Windows **Start** menu.
- Linux: Navigate to <user_home>/ModusToolbox/ide_3.0/eclipse and run the "ModusToolbox" executable.
- macOS: Run the "ModusToolbox.app" executable.

When the Eclipse IDE runs for the first time, a dialog opens to specify the Workspace location. The default location for the workspace is: <user_home>/mtw.

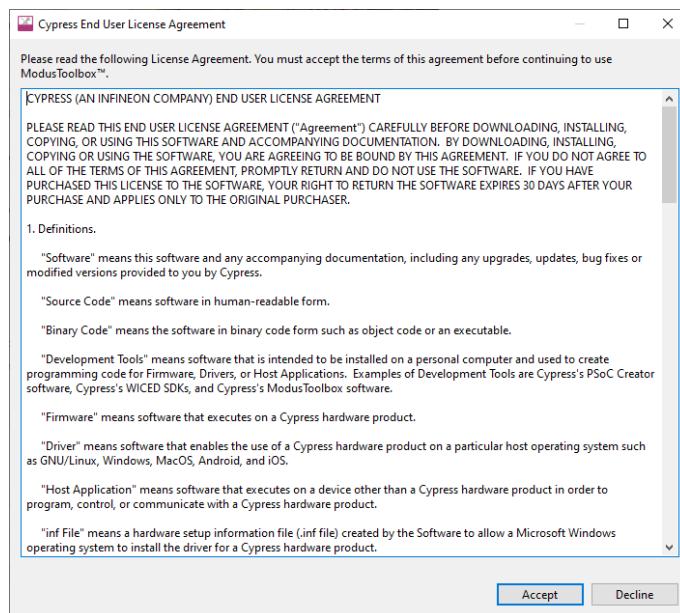


Note: Be aware that the default Eclipse IDE workspace location (<user_home>/mtw) is the same for all versions of ModusToolbox™ software. If you plan to use more than one version, you must specify different workspace names for each one. Enter the workspace location and name and click **Launch** to open the IDE.

If you change the workspace location or name, do not use spaces or illegal characters anywhere in the path.

If your user home directory contains spaces, see [Installing with spaces in user home directory](#).

After the IDE opens for the first time, the End User License Agreement (EULA) displays.



Read the EULA and click **Accept** to proceed. If you click **Decline**, the IDE will close.

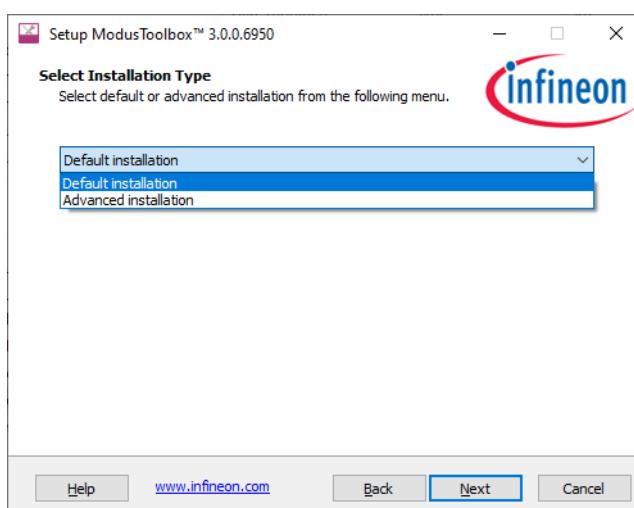
~~3 PSoC™ 6 software tools~~

~~3.1.5 Default versus advanced Windows installation~~

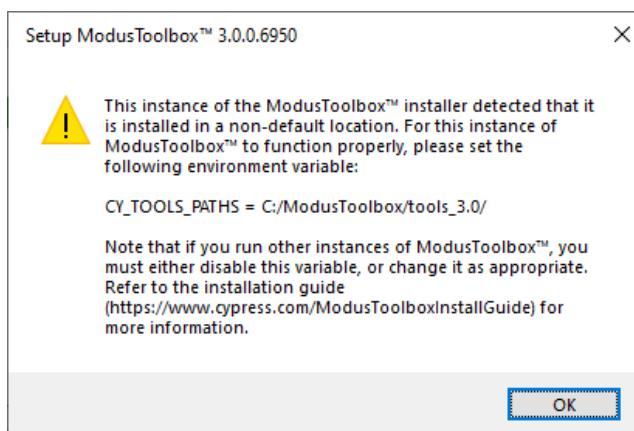
The ModusToolbox™ installer for Windows provides options to install for the current user or for all users of the same computer. Depending on if you have administration privileges or not, you may be asked to enter a password.

Note: If you select "Install for all users" and then later select install for the current user, the Windows "Apps & features" setting will list only one instance of ModusToolbox™ 3.0. Use the uninstaller to point to the type of installation (All Users or Current User), depending on the order they were installed. To see all installations, navigate to **Control Panel > Programs and Features**.

1. After selecting the installation type, follow the prompts to accept the license agreement and select the installation path.
2. On the **Select Installation Type**, choose "Default Installation" or "Advanced Installation."



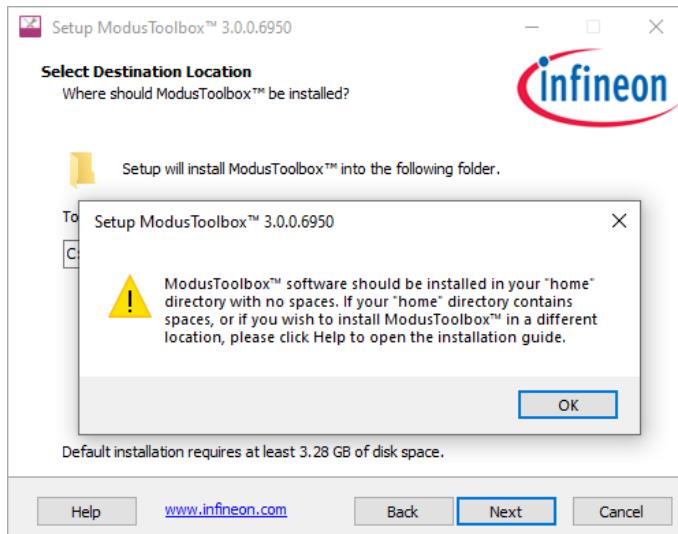
- The default option installs the tools and drivers needed by the ModusToolbox™ tools package.
 - The advanced option allows you to deselect prerequisite software and drivers that are already installed.
3. After continuing with the installation steps and the post-installation process, the following dialog will display if you specified a non-default installation directory as a reminder to set an environment variable.



~~3 PSoC™ 6 software tools~~

~~3.1.6~~ **Installing with spaces in user home directory**

The ModusToolbox™ installer tries to install in your user home directory by default. However, it prevents you from installing into a directory that contains spaces.



If possible, create a new user account and user home directory that doesn't contain spaces. If you cannot create a new user home directory without spaces, then you must perform some extra manual installation steps.

Note: Even though this process is shown for Windows, these steps apply in general to macOS and Linux as well.

3.1.6.1 Step 1: Install at a custom path.

1. Select an alternate installation path that does not include spaces. For example:
C:\MyPath\ModusToolbox
Any path without spaces will work.
2. After installation is complete, create a directory to store your workspaces. For example:
C:\MyPath\mtb-projects
You can choose any path as long as it doesn't contain spaces.
3. Also, create a hidden "dot" directory named ".modustoolbox" to store the cache, offline content, and manifest.loc file discussed later in this section. For example:
C:\MyPath\.modustoolbox

~~3 PSoC™ 6 software tools~~

~~3.1.6.2 Step 2: Create a variable to specify the path to Tools.~~

~~Because you are installing ModusToolbox™ into a non-default location, you need to specify the path to your "tools" directory using an Environment Variable. Open the Environment Variables dialog, and create a new System or User Variable, depending on your installation type (current user or all users). For example:~~

```
CY_TOOLS_PATHS = C:/MyPath/ModusToolbox/tools_3.0
```

Note: Use a Windows-style path (not Cygwin-style, like /cygdrive/c/). Also, use forward slashes.

~~3.1.6.3 Step 3: Create a variable to specify the path to cache.~~

The ModusToolbox™ make system clones all the repos needed for your project, directly into your project. So, the resulting project is self-contained. It uses cache to speed up the clone operations. Normally, the make system would create and use cache directory at:

C:\Users\<user_name>\.modustoolbox\

You need to fix this for the new install location for ModusToolbox™ by changing the location where the make system keeps the cache. Create a new System or User Variable, depending on your installation type (current user or all users). For example:

```
CY_GETLIBS_CACHE_PATH = C:/MyPath/.modustoolbox/cache/
```

Note: Use a Windows-style path (not Cygwin-style, like /cygdrive/c/). Also, use forward slashes.

Alternately, you can disable the caching. The downside is that this will slow down the clone operation and overall project creation, as well as the library update experience. To disable the cache, create a User Variable:

```
CY_GETLIBS_NO_CACHE = 1
```

3 PSoC™ 6 software tools**3.1.6.4 Step 4: Specify the custom path to use for offline content and manifest.loc.**

Although you may not use these features, dependencies require that you set them up while installing the software.

3.1.6.4.1 Offline content path

Specify the non-default location to the "offline" directory with an Environment Variable. For example:

```
CY_GETLIBS_OFFLINE_PATH = C:/MyPath/.modustoolbox/offline/
```

3.1.6.4.2 manifest.loc

Likewise, create an Environment Variable to specify the non-default location of the manifest.loc file. For example:

```
CyManifestLocOverride = C:/MyPath/.modustoolbox/manifest.loc
```

~~3 PSoC™ 6 software tools~~

~~3.1.7 Revision history~~

Revision	Date	Description
**	12/29/2017	New document.
*A	09/18/2018	Complete update for production release.
*B	11/21/2018	Updated the system requirements section. Added information about uninstalling issues. Updated to clarify macOS instructions.
*C	02/27/2019	Updated for version 1.1. Added custom drivers information. Updated linux instructions.
*D	09/26/2019	Added information to clarify usage with multiple versions and workspaces.
*E	10/17/2019	Updated for version 2.0. Added a note for macOS Catalina.
*F	10/21/2019	Added git as a prerequisite.
*G	01/14/2020	Add a link to KBA229345.
*H	02/13/2020	Added a comment about using forward slashes for the CY_TOOLS_PATHS variable.
*I	03/26/2020	Updated for version 2.1.
*J	04/02/2020	Corrected macOS executable name.
*K	04/14/2020	Corrected "optional" step for installing with spaces in user home directory.
*L	09/01/2020	Updated for version 2.2. Updated to include Python 3.7 requirement. Removed macOS Catalina notarization warning.
*M	03/25/2021	Updated for version 2.3. Added installer instructions for Windows and multiple users. Added Linux instruction for libncurses5. Updated for macOS Big Sur.
*N	9/10/2021	Updated for version 2.4.
*O	4/12/2021	Updated for version 3.0.

3.2 Eclipse IDE for ModusToolbox™

About this document

-
- 2

Scope and purpose

ModusToolbox™ software is a set of tools that support device configuration and application development. These tools enable you to integrate our devices into your existing development methodology.

Intended audience

This guide provides a quick walkthrough for using the Eclipse IDE provided as part of the ModusToolbox™ tools package. If you wish to use other IDEs, such as VS Code, IAR, or µVision, refer to the "Export to IDEs" chapter in the [ModusToolbox™ user guide](#).

~~3 PSoC™ 6 software tools~~

~~Reference documents~~

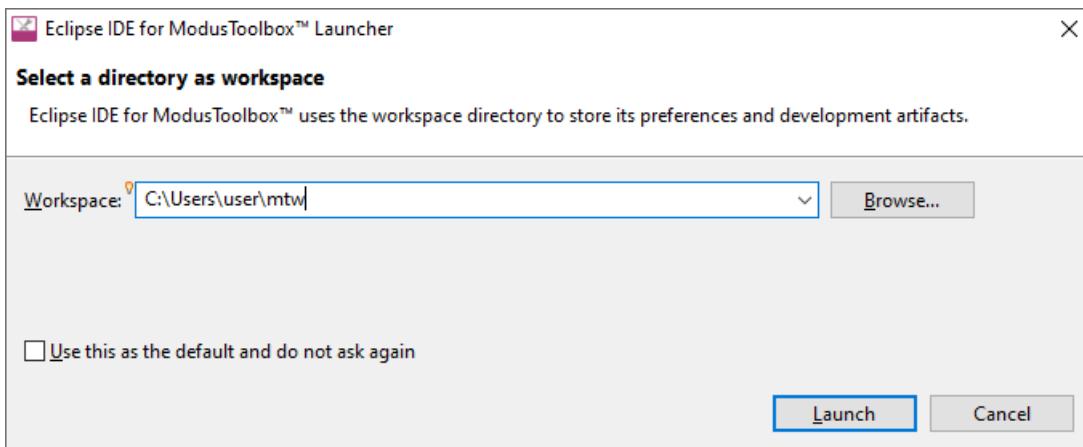
Refer to the following documents for more information as needed:

- [ModusToolbox™ installation guide](#) – This provides information and instructions about installing the tools package on Windows, Linux, and macOS.
- [Eclipse IDE for ModusToolbox™ user guide](#) – This provides more details about features added to the ModusToolbox™ version of the Eclipse IDE.
- [Eclipse IDE survival guide](#) – This provides a FAQ about how to use Eclipse to perform common tasks.
- [Eclipse Workbench User Guide](#) – This provides more details about the generic Eclipse environment.
- [ModusToolbox™ user guide](#) – This provides information about all the tools included with ModusToolbox™.
- [Project Creator user guide](#) – This provides specific information about the Project Creator tool.
- [Device Configurator guide](#) – This provides specific information about the Device Configurator.

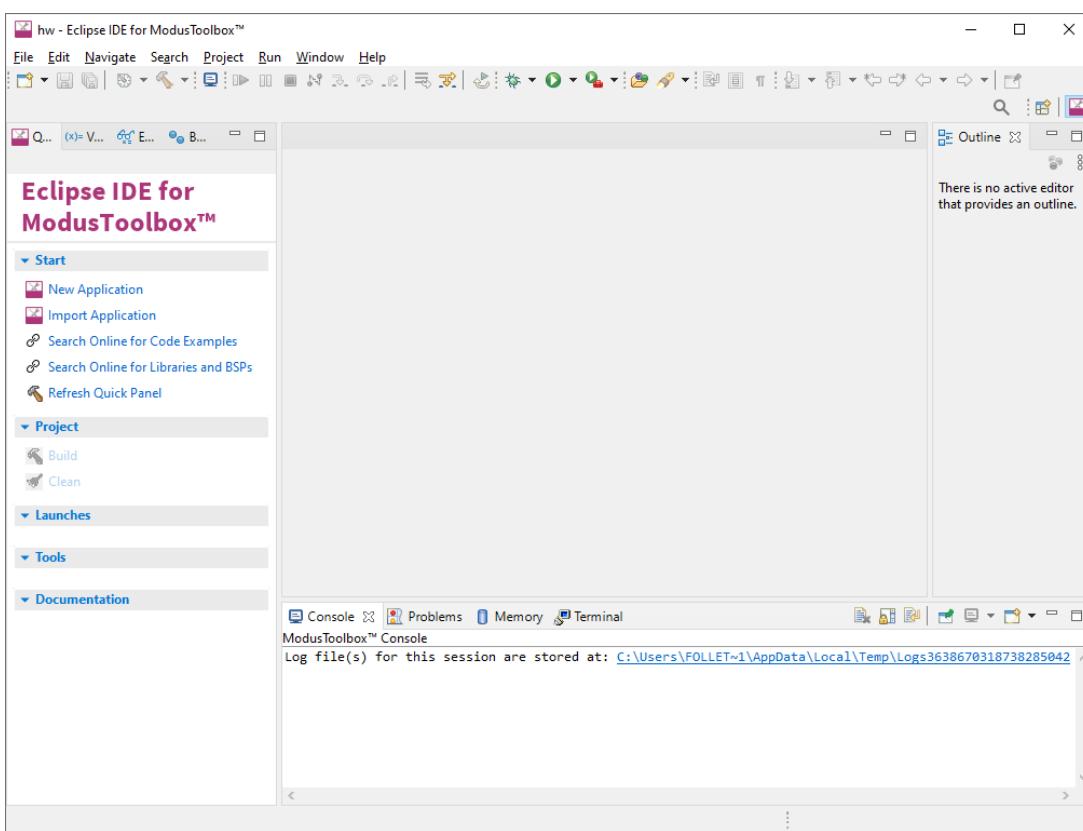
~~3 PSoC™ 6 software tools~~

~~3.2.1 Download/Install/Run Eclipse IDE~~

Refer to the instructions in the [ModusToolbox™ installation guide](#) for how to download and install ModusToolbox™ software, as well as to run the Eclipse IDE. As noted in the guide, when you first run the IDE, a dialog opens to specify the workspace.



Then, accept the License Agreement, and the Eclipse IDE opens with the Quick Panel and Console by default.



- The Quick Panel provides links to tool commands and documentation.
- The Console displays various messages when updating the application in some way (building, programming, etc.).

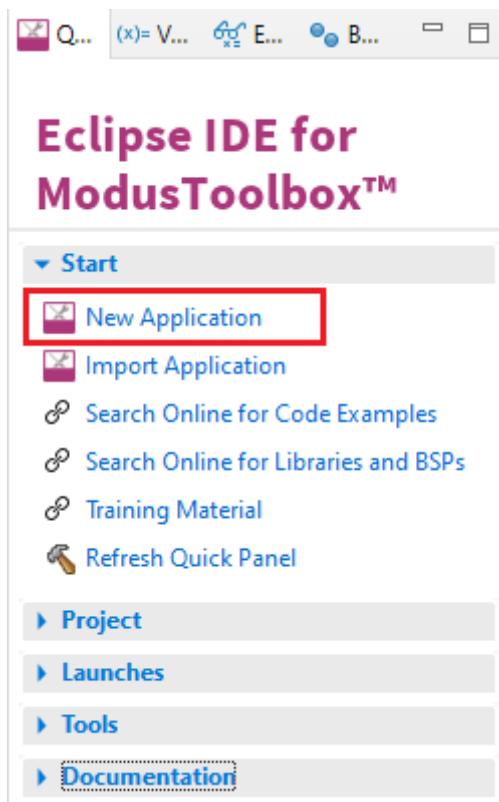
3 PSoC™ 6 software tools

3.2.2 Create new application

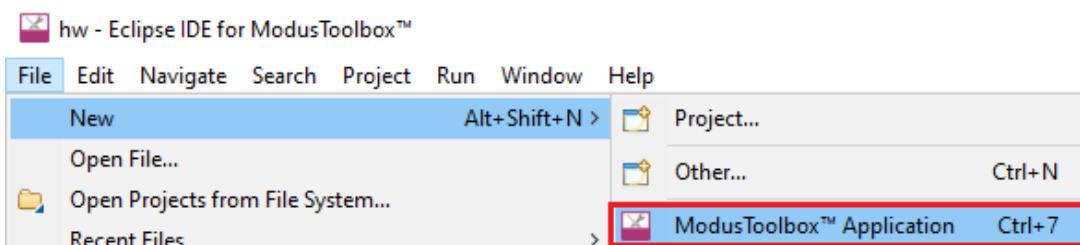
Creating an application includes several steps, as follows:

3.2.2.1 Step 1: Open Project Creator tool

In the Quick Panel, click the New Application link.

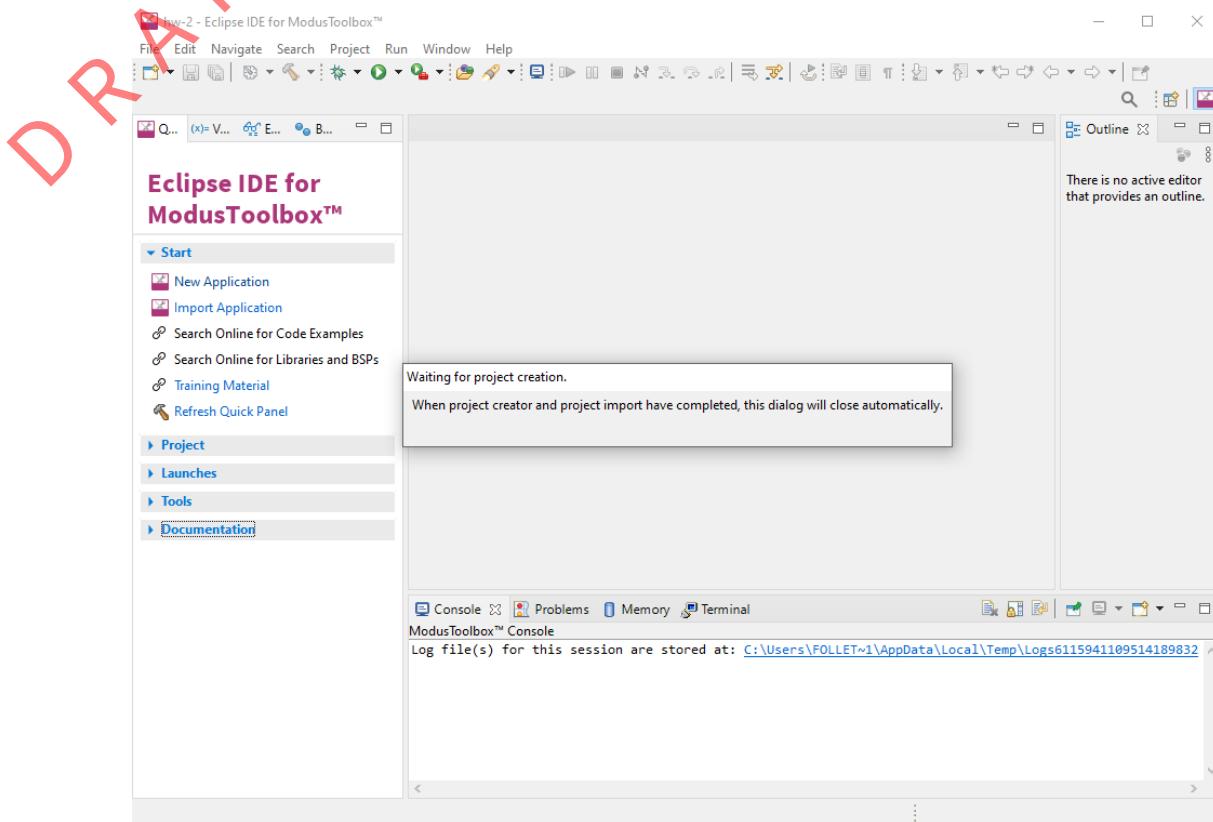


Or, select **File > New > ModusToolbox™ Application**.



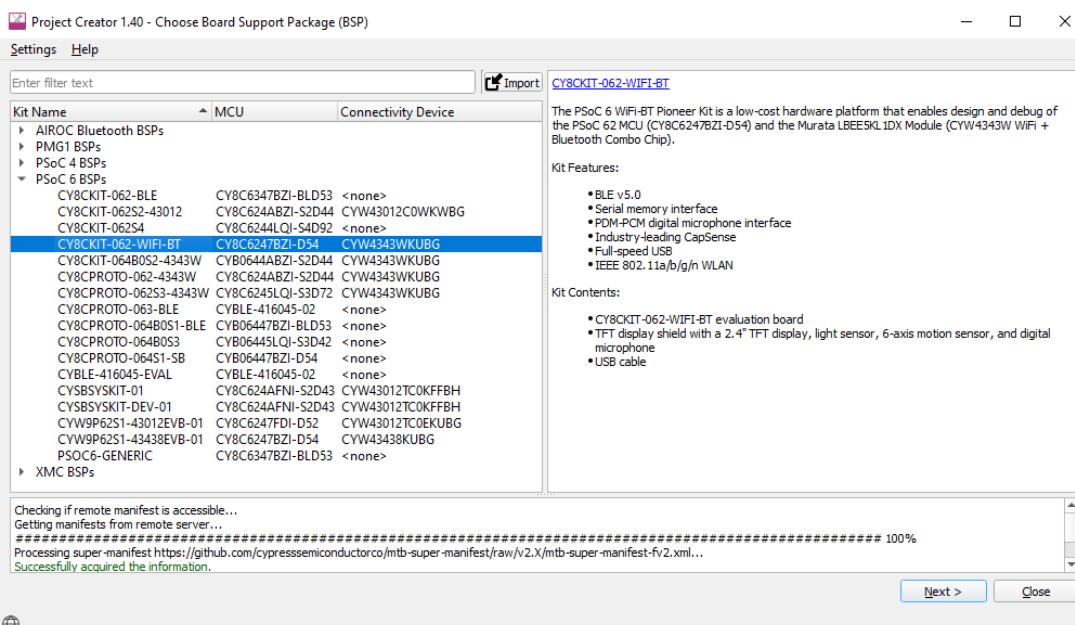
The Eclipse IDE displays a message and waits for the Project Creator tool to open.

3 PSoC™ 6 software tools



3.2.2.2 Step 2: Choose Board Support Package (BSP)

When the Project Creator tool opens, click on the Kit Name; see the description for it on the right. For this example, select the CY8CKIT-062-WIFI-BT kit. The following image is an example; the precise list of boards available in this version will reflect the platforms available for development.

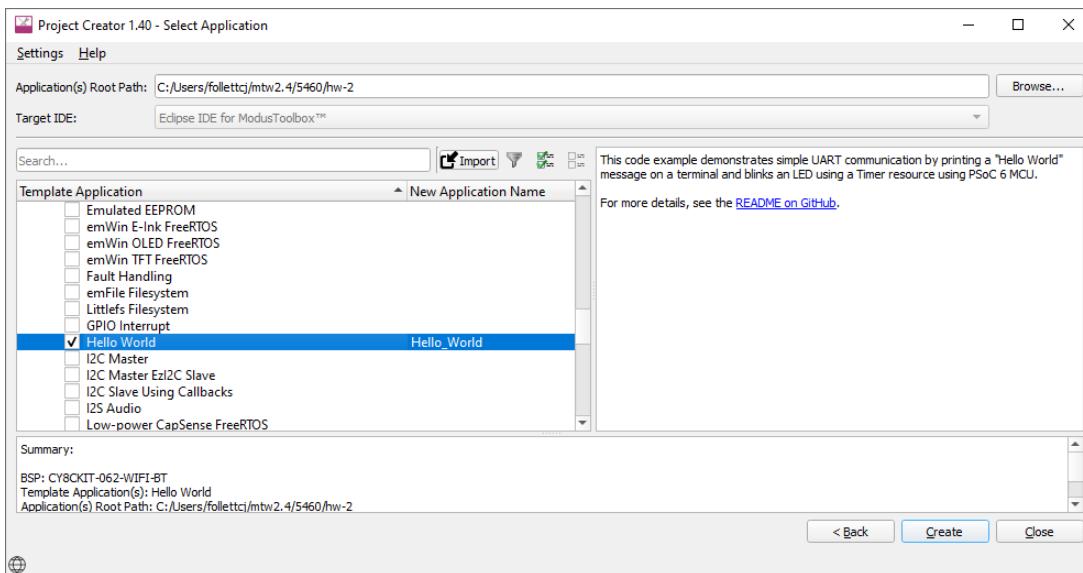


~~3 PSoC™ 6 software tools~~

~~3.2.2.3 Step 3: Select application~~

Click **Next >** to open the Select Application page. This page displays example applications, which demonstrate different features available on the selected BSP. In this case, the CY8CKIT-062-WIFI-BT provides the PSoC™ 62 MCU and the AIROC™ CYW4343W Wi-Fi & Bluetooth® combo chip. You can create examples for PSoC™ 6 MCU resources such as CAPSENSE™ and QSPI, as well as numerous examples for connectivity.

For this example, select Hello World from the list. This example exercises the PSoC™ 6 MCU to blink an LED.



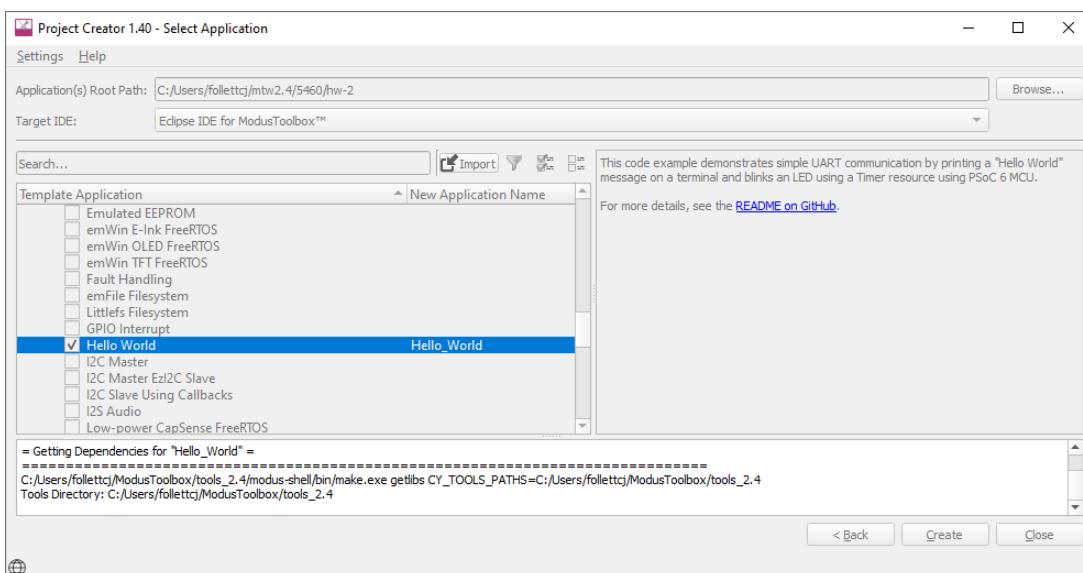
Note: The actual application names available might vary.

Type a name for your application or leave the default name. Do not use spaces in the application name. Also, do not use common illegal characters, such as:

* . " ' / \ [] : ; | = ,

~~3.2.2.4 Step 4: Create application~~

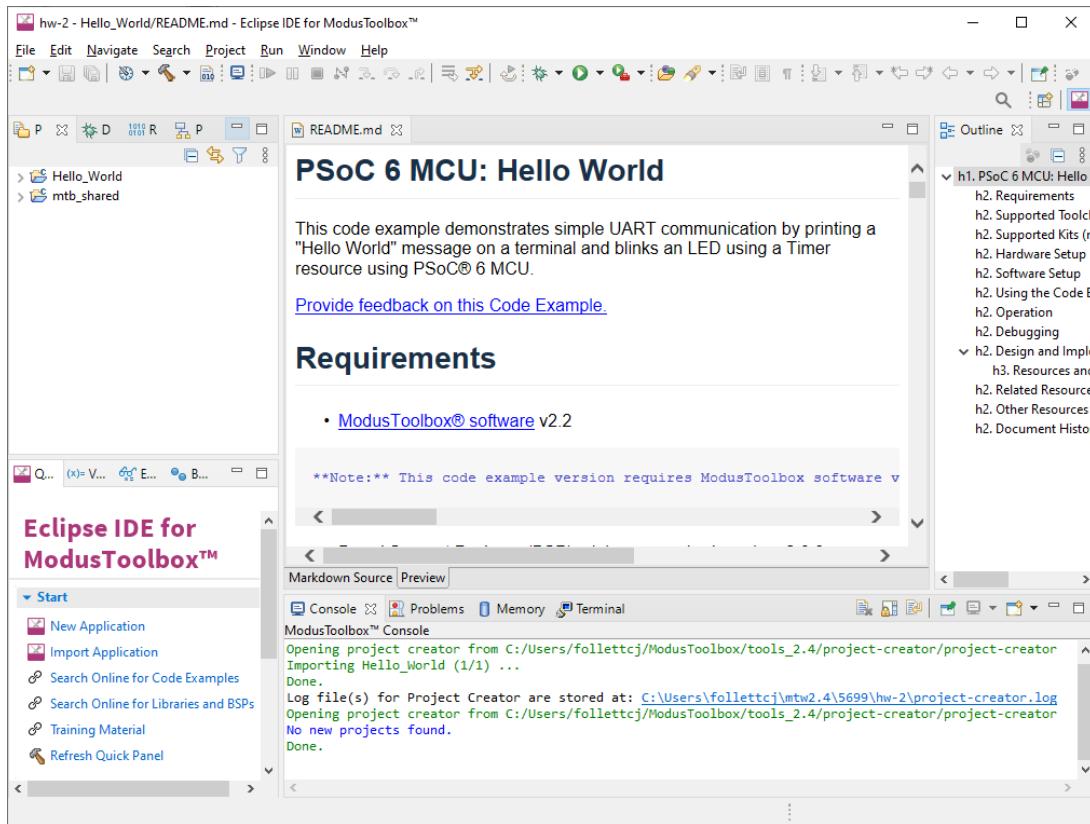
Click **Create** to start creating the application. The tool displays various messages.



When the process completes, a message states that the application was created, and the Project Creator tool closes automatically. For more information, refer to the [Project Creator user guide](#).

3 PSoC™ 6 software tools

After several moments, the application opens in the Eclipse IDE with the Hello_World project in the Project Explorer, with various files and directories. The README.md file opens in the file viewer/code editor where you can read the text.

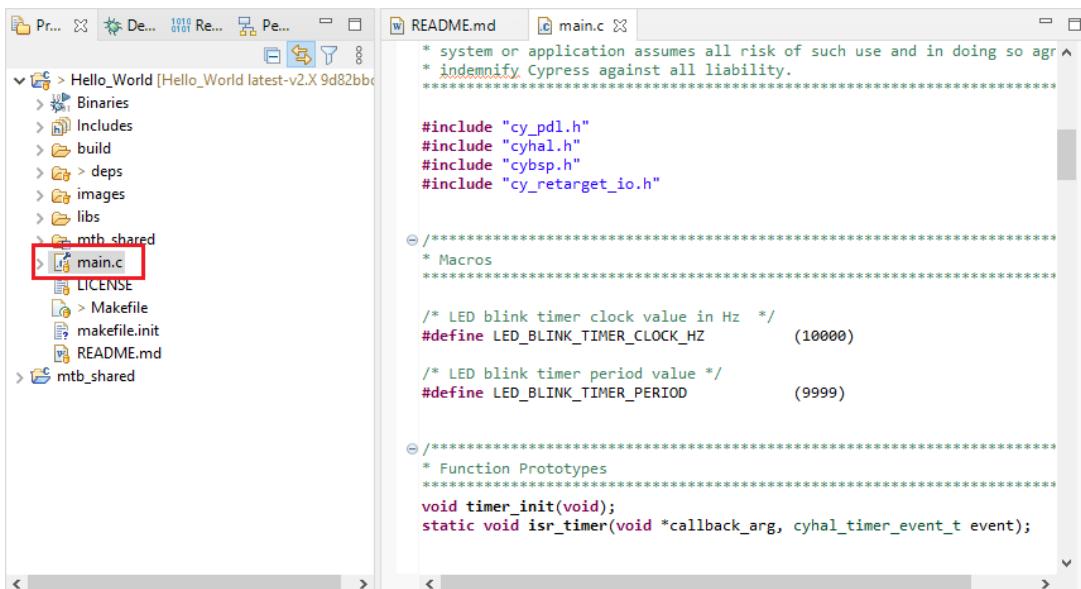


3 PSoC™ 6 software tools

3.2.3 Add/modify application code

Code example applications work as they are, and there is no need to add or modify code in order to build or program them. However, if you want to update and change the application to do something else, open the appropriate file in the code editor.

Double-click the main.c file to open it.



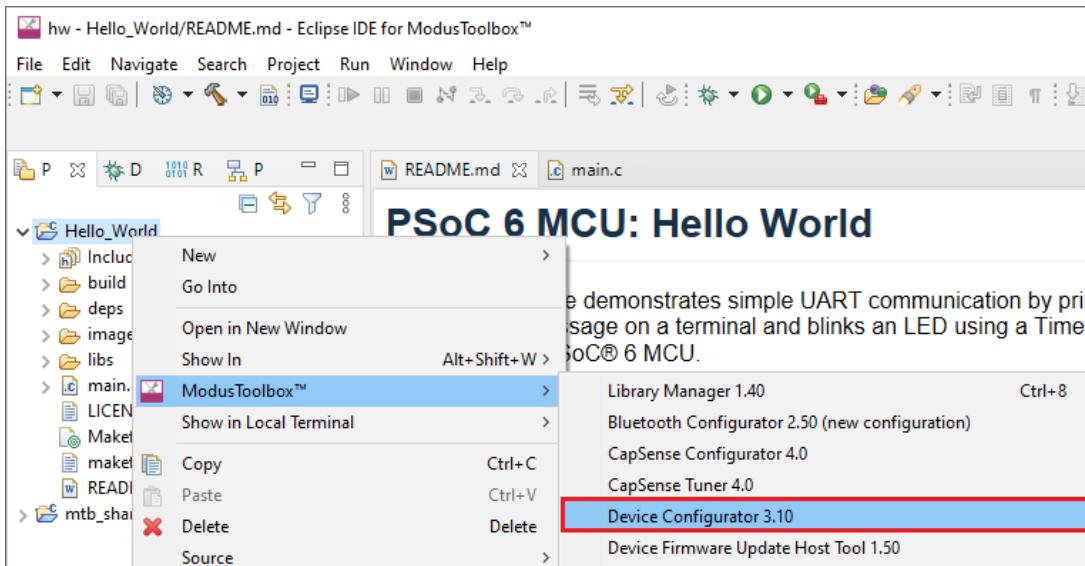
```
* system or application assumes all risk of such use and in doing so agrees to indemnify Cypress against all liability.  
*****  
#include "cy_pdl.h"  
#include "cyhal.h"  
#include "cybsp.h"  
#include "cy_retarget_io.h"  
  
/* Macros  
*****  
/* LED blink timer clock value in Hz */  
#define LED_BLINK_TIMER_CLOCK_HZ (10000)  
  
/* LED blink timer period value */  
#define LED_BLINK_TIMER_PERIOD (9999)  
  
/* Function Prototypes  
*****  
void timer_init(void);  
static void isr_timer(void *callback_arg, cyhal_timer_event_t event);
```

As you type into the file, an asterisk (*) will appear in the file's tab to indicate changes were made. The Save/Save As commands will also become available to select.

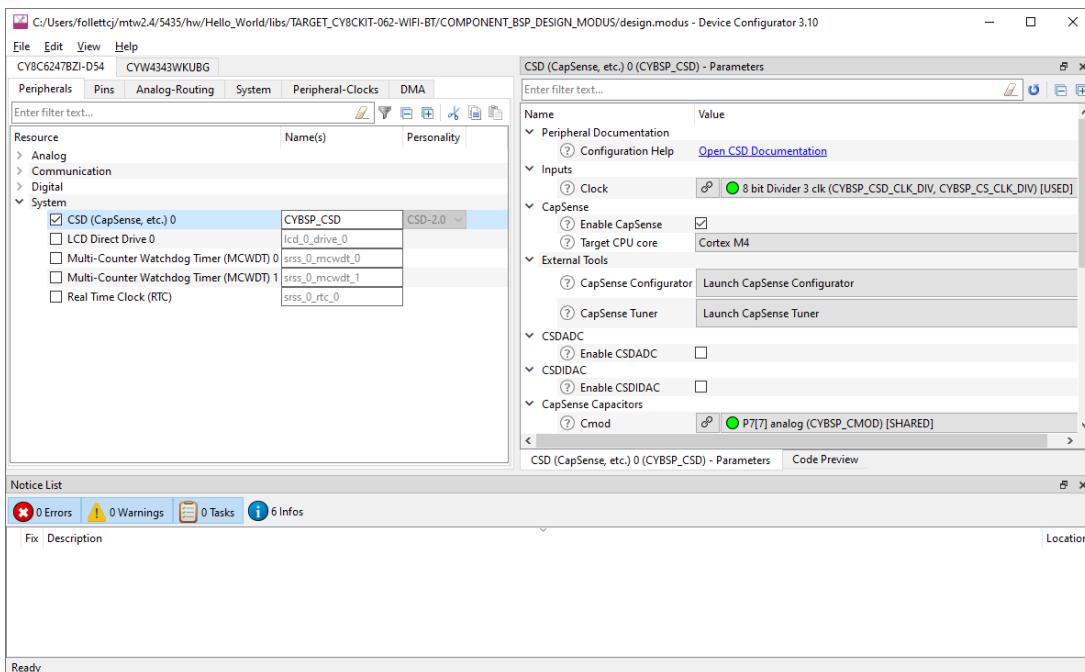
~~3 PSoC™ 6 software tools~~

~~3.2.4 View device resources~~

To view peripherals, pins, clocks, etc., open the Device Configurator. Right-click on the project folder and select **ModusToolbox™ Device Configurator <version>**.



The Device Configurator provides access to the BSP resources and settings. Each enabled resource contains one or more links to the related API documentation. There are also buttons to open other configurators for CAPSENSE™, QSPI, Smart I/O, etc. For more information, refer to the [Device Configurator guide](#), which is also available by selecting **View Help** from the tool's **Help** menu.



Note: If you make changes to settings in any of these configurators, you are making changes to a standard BSP library, which will cause the repo to become dirty. Additionally, if the BSP is in the shared asset repository, changes will impact all applications that use the shared BSP. If you wish to make changes, you should first copy the configuration information to the application and override the BSP configuration or create a custom BSP. For more information, refer to the [ModusToolbox™ user guide](#).

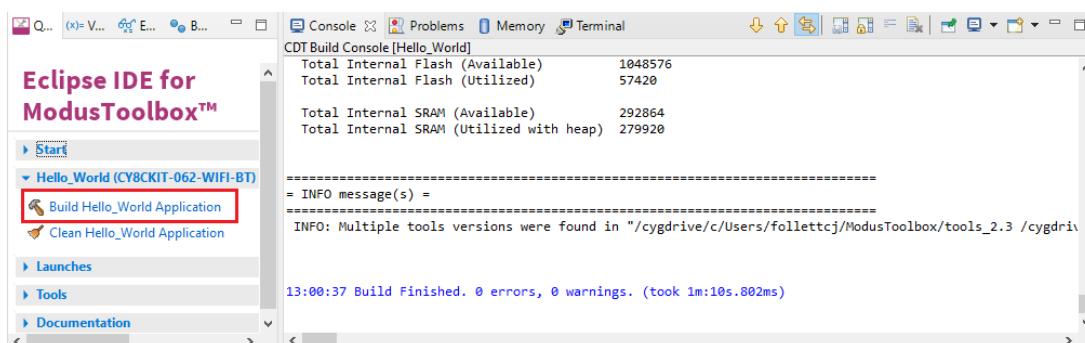
Note: The Device Configurator cannot be used to open Library Configurators, such as Bluetooth® and USB.

~~3 PSoC™ 6 software tools~~

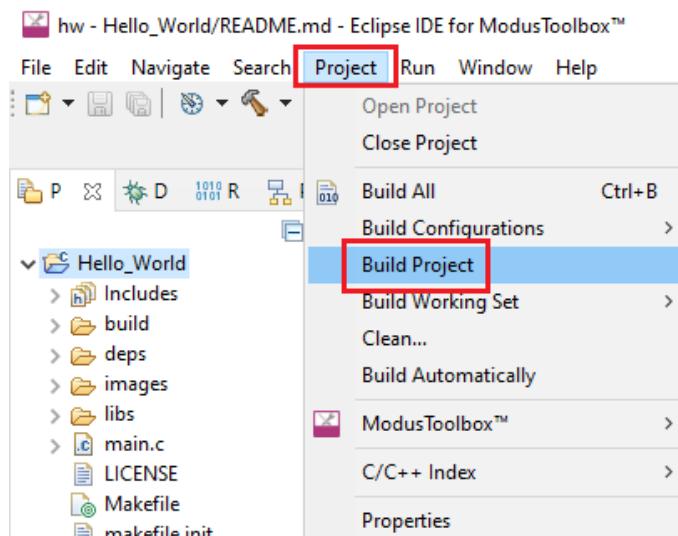
~~3.2.5 Build the Application~~

Building the application is not specifically required, because building will be performed as part of the programming and debugging process. However, if you are running the Eclipse IDE without any hardware attached, you may wish to build your application to ensure all the code is correct.

In the Project Explorer, click on the Hello_World project. In the Quick Panel, click the **Build Hello_World Application** link. Build information will display in the Console pane.



If you have the Hello_World project selected, you can also select **Build Project** from the **Project** menu or from the right-click menu. The Eclipse menus change based on what you have selected in the Project Explorer.



3 PSoC™ 6 software tools

3.2.6 Program the Device

There are several different "Launch Configurations" for programming and debugging the various development kits and starter applications within the Eclipse IDE. These Launch Configurations provide details about how to configure the application. This section provides a brief walkthrough for programming this example. For more details about other Launch Configurations and settings, refer to the [Eclipse IDE for ModusToolbox™ user guide](#), "Program and Debug" chapter.

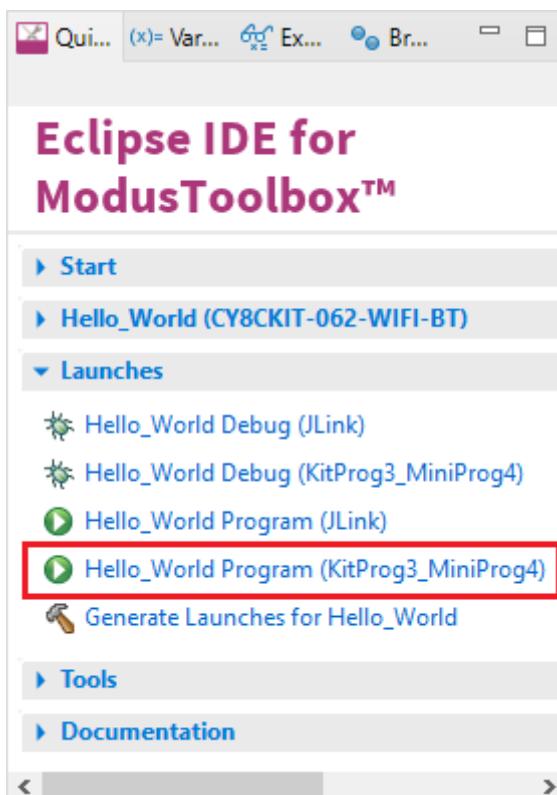
3.2.6.1 Connect the Kit

Follow the instructions provided with the CY8CKIT-062-WIFI-BT kit to connect it to the PC with the USB cable.

Note: This kit's firmware may be set to KitProg2. To use this kit with the ModusToolbox™ software, you must upgrade the firmware to KitProg3. Refer to the [KitProg3 user guide](#) for more details.

3.2.6.2 Program

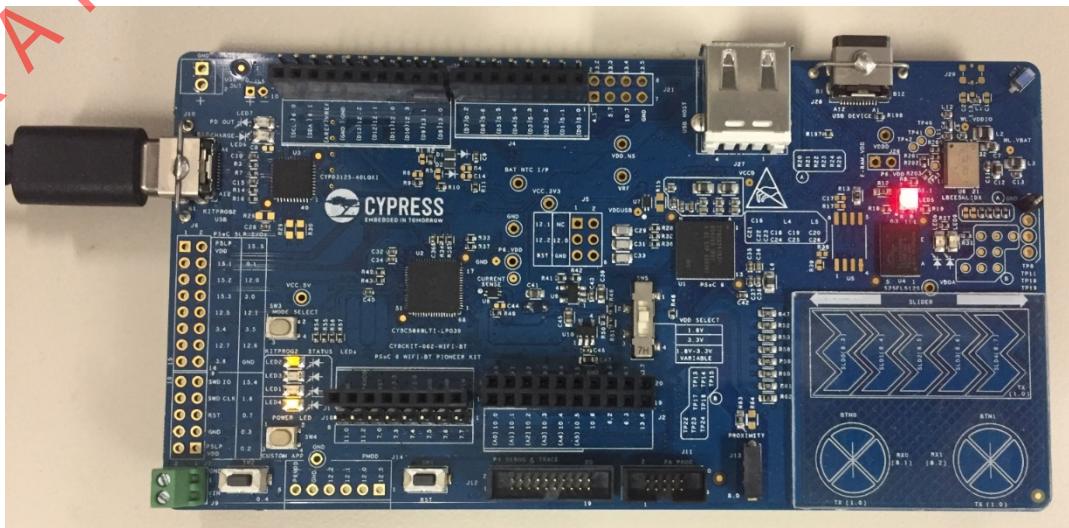
In the Project Explorer, select the Hello_World application. Then, in the Quick Panel, click the **Hello_World Program (KitProg3_MiniProg4)** link.



If needed, the IDE builds the application and messages display in the Console. If the build is successful, device programming starts immediately. If there are build errors, then error messages will indicate as such. When complete, the LED will start blinking.

3 PSoC™ 6 software tools

DRAFT

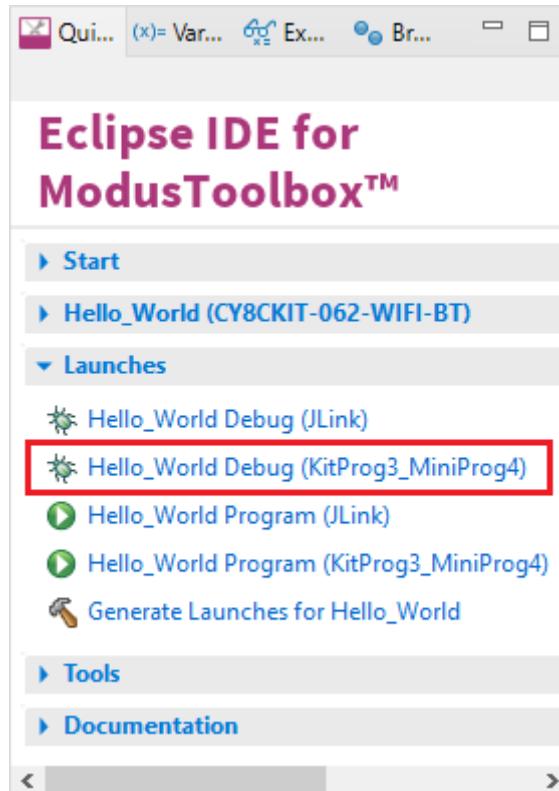


3 PSoC™ 6 software tools

3.2.7 ~~DRAFT~~ Debug the program

As mentioned under the [Program the Device](#) section in this document, there are many different Launch Configurations for different kits and applications. This section provides a brief walkthrough for debugging this example. For more details about other Launch Configurations and settings, refer to the [Eclipse IDE for ModusToolbox™ user guide](#), "Program and Debug" chapter.

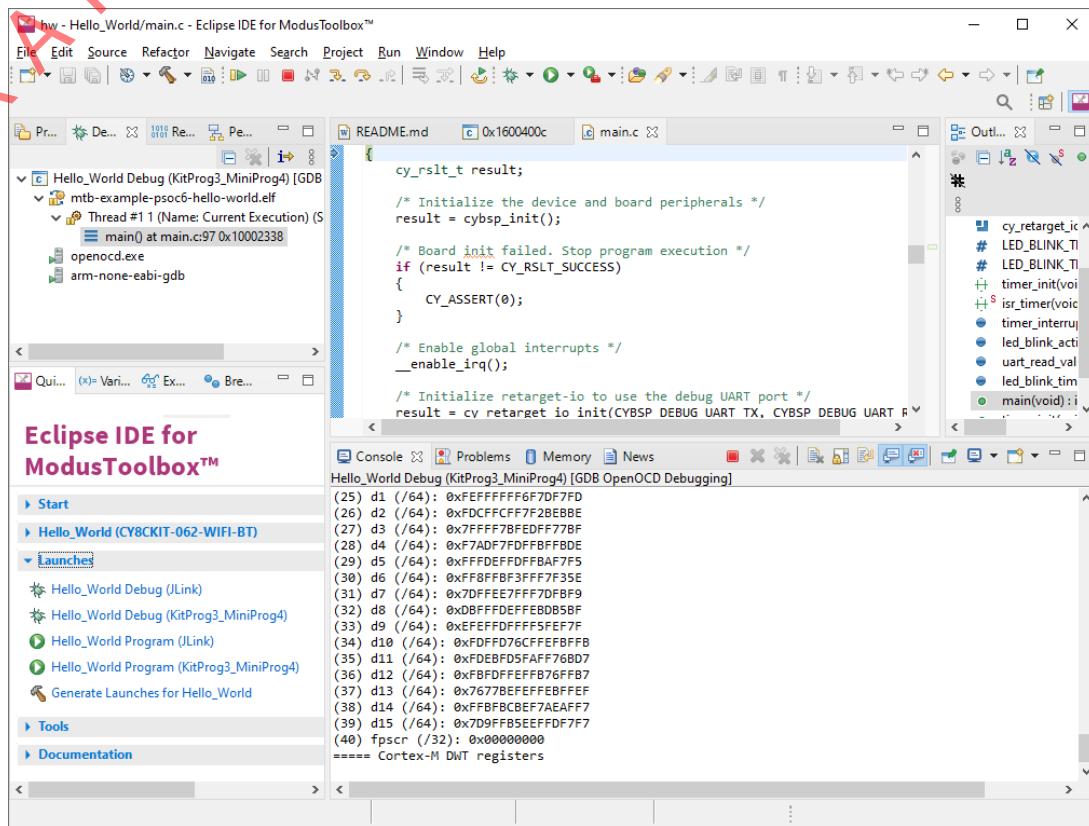
In the Project Explorer, select the Hello_World application. Then, in the Quick Panel, click the **Hello_World Debug (KitProg3)** link under Launches.



If needed, the IDE builds the application and messages display in the Console. If the build is successful, the IDE switches to debug mode automatically. If there are build errors, then error messages will indicate as such.

3 PSoC™ 6 software tools

A screenshot of the Eclipse IDE interface. The title bar says "hw - Hello_World". The menu bar includes "File", "Edit", and "Source". Below the menu is a toolbar with icons for file operations. A central view shows a file tree with "HelloWorld.java" and "HelloWorld.class" under the project root. A status bar at the bottom shows "Hello_World" and "1 file(s) selected". A large red "DRAFT" watermark is diagonally across the screen.



~~3 PSoC™ 6 software tools~~

~~3.2.8 Revision history~~

Date	Revision	Description
12/29/2017	**	New document.
11/20/2018	*A	Completely rewritten from previous version based on Eclipse changes.
03/05/2019	*B	Updated screen captures and associated text for version 1.1.
10/16/2019	*C	Updated for version 2.0.
03/26/2020	*D	Updated for version 2.1.
04/02/2020	*E	Fixed broken links.
06/22/2020	*F	Updated screen captures to show AnyCloud.
09/01/2020	*G	Updated for version 2.2.
03/25/2021	*H	Updated for version 2.3; includes new version of Eclipse.
09/13/2021	*I	Updated for version 2.4.

3.3 ModusToolbox™ user guide

About this document

-
- 3

Scope and purpose

This guide provides information and instructions for using the ModusToolbox™ tools provided by the version 3.0 installer and the make build system. This document contains the following chapters:

- Chapter 1 describes ModusToolbox™ software.
- Chapter 2 provides instructions for getting started using the ModusToolbox™ tools.
- Chapter 3 describes the ModusToolbox™ build system.
- Chapter 4 covers different aspects of the ModusToolbox™ board support packages (BSPs).
- Chapter 5 explains the ModusToolbox™ manifest files and how to use them with BSPs, libraries, and code examples.
- Chapter 6 provides instructions for using a ModusToolbox™ application with various integrated development environments (IDEs).

Intended audience

This document helps application developers understand how to use all the tools included with ModusToolbox™ software.

Abbreviations and definitions

The following define the abbreviations and terms used in this document that you may not be familiar with:

- BSP – board support package
- PDL – peripheral driver library
- HAL – hardware abstraction layer
- WHD – Wi-Fi host driver
- WCM – Wi-Fi connection manager

~~3 PSoC™ 6 software tools~~

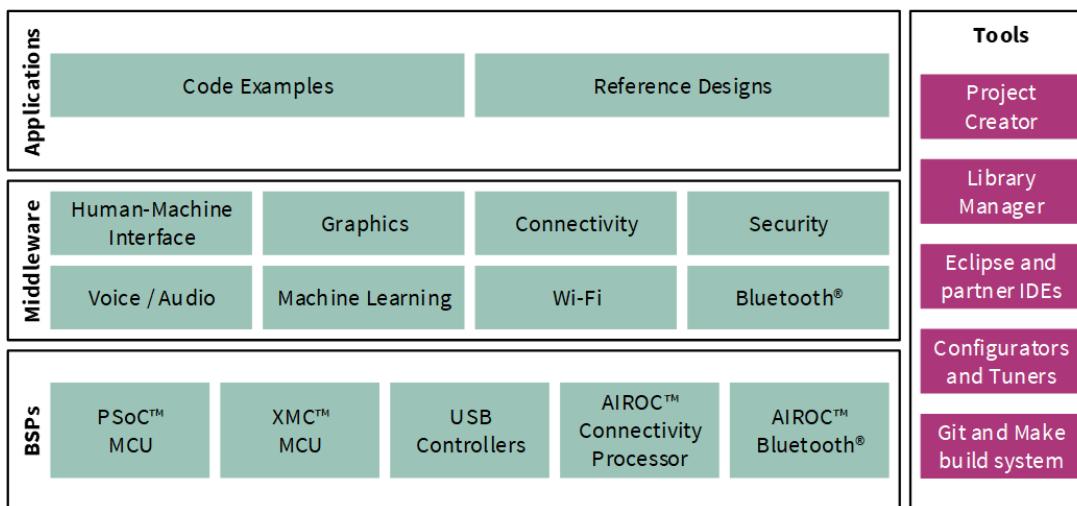
~~3.3.1~~ Introduction

This chapter provides an overview of the ModusToolbox™ software environment, which provides support for many types of devices and ecosystems.

~~3.3.1.1~~ What is ModusToolbox™ software?

ModusToolbox™ software is a modern, extensible development environment supporting a wide range of Infineon microcontroller devices. It provides a flexible set of tools and a diverse, high-quality collection of application-focused software. These include configuration tools, low-level drivers, libraries, and operating system support, most of which are compatible with Linux-, macOS-, and Windows-hosted environments.

The following diagram shows a very high-level view of what is available as part of ModusToolbox™ software. This is not a comprehensive list. It merely conveys that there are multiple resources available to you.



ModusToolbox™ software does not include proprietary tools or custom build environments. This means you choose your compiler, your IDE, your RTOS, and your ecosystem without compromising usability or access to our industry-leading CAPSENSE™, AIROC™ Wi-Fi and Bluetooth®, security, and various other features.

Another important aspect of the ModusToolbox™ software is that each product is versioned. This ensures that each product can be updated on an ongoing basis, but it also allows you to lock down specific versions of the tools for your specific environment. See [Product versioning](#) for more details.

3.3.1.2 Run-time software

ModusToolbox™ tools also include an extensive collection of [GitHub-hosted repos](#) comprising Code Examples, BSPs, plus middleware and applications support. We release run-time software on a quarterly "train model" schedule, and access to new or updated libraries typically does not require you to update your ModusToolbox™ installation.

New projects start with one of our many [Code examples](#) that showcase everything from simple peripheral demonstrations to complete application solutions. Every Infineon kit is backed by a comprehensive BSP implementation that simplifies the software interface to the board, enables applications to be re-targeted to new hardware in no time, and can be easily extended to support your custom hardware without the usual porting and integration hassle.

The extensive middleware collection includes an ever-growing set of sensor interfaces, display support, and connectivity-focused libraries. The ModusToolbox™ installer also conveniently bundles packages of all the necessary run-time components you need to leverage the following key Infineon technology focus areas:

- CAPSENSE™ technology
- AnyCloud (AIROC™ Wi-Fi and Bluetooth® applications)

~~3 PSoC™ 6 software tools~~

- Machine Learning
- Device Security (PSoC™ 64 "Secure Boot" MCU)

~~3.3.1.2.1 Code examples~~

All current ModusToolbox™ examples can be found through the GitHub [code example page](#). There you will find links to examples for the Bluetooth® SDK, PSoC™ 6 MCU, PSoC™ 4 device, among others. For most code examples examples, you can use [git clone](#) or the [Project Creator tools](#) to create an application and use it directly with ModusToolbox™ tools. For some examples, like Mbed OS, you will need to follow the directions in the code example repository to instantiate the example. Instructions vary based on the nature of the application and the targeted ecosystem.

In the ModusToolbox™ build infrastructure, any example application that requires a library downloads that library automatically.

You can control the versions of the libraries being downloaded and also their location on disk, and whether they are shared or local to the application. Refer to the [Library Manager user guide](#) for more details.

3.3.1.2.2 Libraries (middleware)

In addition to the code examples, there are many other parts of ModusToolbox™ that are provided as libraries. These libraries are essential for taking full advantage of the various features of the various devices. When you create a ModusToolbox™ application, the system downloads all the libraries your application needs. See [ModusToolbox™ build system](#) chapter to understand how all this works.

All current ModusToolbox™ libraries can be found through the GitHub [ModusToolbox™ software page](#). A ModusToolbox™ application can use different libraries based on the Active BSP. In general, there are several categories of libraries. Each library is delivered in its own repository, complete with documentation.

Common library types:

Most BSPs have some form of the following types of libraries:

- Abstraction Layers – This is usually the RTOS Abstraction Layer.
- Base Libraries – These are core libraries, such as core-lib and core-make.
- Board Utilities – These are board-specific utilities, such as display support or BTSp.
- MCU Middleware – These include MCU-specific libraries such as freeRTOS or Clib support.

AIROC™ Bluetooth® Libraries:

For the AIROC™ Bluetooth® BSPs, there specific libraries that do not apply to any other BSPs, including:

- BTSDK Chip Libraries
- BTSDK Core Support
- BTSDK Shared Source Libraries
- BTSDK Utilities and Host/Peer Apps

BSP-specific base libraries:

BSP-specific libraries include mtb-hal, mtb-pdl, and recipe-make. Some of these are identified as device-specific using the following categories:

- cat1/cat1a = PSoC™ 6 MCUs (mtb-hal-cat1, recipe-make-cat1a, etc.)
- cat2 = PSoC™ 4 devices and XMC™ Industrial MCUs (mtb-hal-cat2, mtb-pdl-cat2)
- cat3 = XMC™ Industrial MCUs (recipe-make-cat3)

~~3 PSoC™ 6 software tools~~

~~PSoC™ 6 additional libraries:~~

Due to the nature of the PSoC™ 6 MCU, plus the combo devices, certain PSoC™ 6 BSPs have additional libraries, including:

- Bluetooth® Middleware Libraries – These are for the BTStack and Bluetooth® FreeRTOS.
- PSoC™ 6 Middleware – These are libraries specific to the PSoC™ 6 MCU, such as EMEEPROM and DFU.
- Wi-Fi Middleware Libraries – These are libraries for AnyCloud applications on a PSoC™ 6 MCU with AIROC™ CYW43xxx Wi-Fi & Bluetooth® combo chip.

3.3.1.2.3 BSPs

The BSP is a central feature of ModusToolbox™ software. The BSP specifies several critical items for the application, including:

- hardware configuration files for the device (for example, design.modus)
- startup code and linker files for the device
- other libraries that are required to support a kit

BSPs are aligned with our development/evaluation kits; they provide files for basic device functionality. A BSP typically has a design.modus file that configures clocks and other board-specific capabilities. That file is used by the ModusToolbox™ configurators. A BSP also includes the required device support code for the device on the board. You can modify the configuration to suit your application.

Supported devices

ModusToolbox™ software supports development on the following Arm Cortex-M devices.

- AIROC™ Wi-Fi and Bluetooth® chips
- PMG1 USB-C Power Delivery Microcontroller
- PSoC™ 4 Configurable Microcontroller (See [AN79953: Getting Started with PSoC™ 4](#) for the supported PSoC™ 4 devices.)
- PSoC™ 6 MCU
- PSoC™ 64 "Secure Boot" MCU
- XMC™ Industrial Microcontroller

BSP releases

We release BSPs independently of ModusToolbox™ software as a whole. This [search link](#) finds all currently available BSPs on our GitHub site.

The search results include links to each repository, named TARGET_kit-number. For example, you will find links to repositories like [TARGET_CY8CPROTO-062-4343W](#). Each repository provides links to relevant documentation. The following links use this BSP as an example. Each BSP has its own documentation.

The information provided varies, but typically includes one or more of:

- An [API reference for the BSP](#)
- The [BSP overview](#)
- a link to the [associated kit page](#) with kit-specific documentation

A BSP is specific to a board and the device on that board. For custom development, you can create or modify a BSP for your device. See the [Board support packages](#) chapter for how they work and how to create your own for a custom board.

~~3 PSoC™ 6 software tools~~

~~3.3.1.3 Development tools~~

The ModusToolbox™ tools package provides you with all the desktop products needed to build sophisticated, low-power embedded, connected and IoT applications. The tools enable you to create new applications (Project Creator), add or update software components (Library Manager), set up peripherals and middleware (Configurators), program and debug (OpenOCD and Device Firmware Updater), and compile (GNU C compiler).

Infineon Technologies understands that you want to pick and choose the tools and products to use, merge them into your own flows, and develop applications in ways we cannot predict. That's why ModusToolbox™ software is not a monolithic, proprietary software tool that dictates the use of any particular IDE.

For convenience, the tools package installation includes the Eclipse IDE for ModusToolbox™. However, we fully support the following IDEs and their corresponding compiler technology, so you are free to develop the way you wish:

- Microsoft Visual Studio Code (VS Code)
- IAR Embedded Workbench (EW-ARM)
- Arm Microcontroller Developers Kit (μVision 5)

For detailed instructions developing ModusToolbox™ applications with third-party IDEs, see the [Exporting to supported IDEs](#) chapter in this guide.

The [ModusToolbox™ tools package installer](#) provides required and optional core resources for any application.

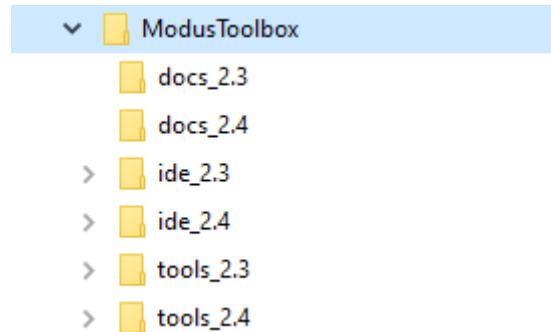
This section provides an overview of the available resources:

- [Directory structure](#)
- [Documentation](#)
- [IDE support](#)
- [Tools](#)

The installer does not include [Code examples](#) or [Libraries \(middleware\)](#), but it does provide the tools to access them.

3.3.1.3.1 Directory structure

Refer to the [ModusToolbox™ installation guide](#) for information about installing ModusToolbox™. Once it is installed, the various ModusToolbox™ top-level directories are organized as follows:



Note: This image shows ModusToolbox™ versions 2.3 and 2.4 installed. Your installation may only include ModusToolbox™ version 2.4. Refer to the [Product versioning](#) section for more details.

The ModusToolbox directory contains the following subdirectories for version 2.4:

- docs_2.4 – This is the top-level documentation directory. It contains various top-level documents and an html file with links to documents provided as part of ModusToolbox™ software. See [Documentation](#) for more information.

~~3 PSoC™ 6 software tools~~

- ide_2.4:
 - eclipse (or ModusToolbox.app on macOS) – This contains the optional Eclipse IDE for ModusToolbox™. It includes the ModusToolbox™ perspective, application management, code authoring and editing, build tools, and debug capabilities. The IDE supports the C and C++ programming languages. It includes the GCC Arm build tools. It supports debugging via OpenOCD or J-Link. For more details, refer to the [Eclipse IDE for ModusToolbox™ software user guide](#).
- tools_2.4: This contains all the various tools and scripts installed as part of ModusToolbox™. See [Tools](#) for more information.

3.3.1.3.2 Documentation

The docs directory contains top-level documents and an HTML document with links to all the documents included in the installation and on the web.

Release notes

For the 2.4 release, the release notes document is for all of the ModusToolbox™ software included in the installation.

Top-level documents

This folder contains the Eclipse IDE documentation, the ModusToolbox™ software installation guide, and this user guide. These guides cover different aspects of using the IDE and various ModusToolbox™ tools.

Document index page

The doc_landing.html file provides links to all the documents included in the installation and on the web. This file is also available from the IDE Help menu.

ModusToolbox™ 2.4 documentation

This page provides brief descriptions and links to various types of documentation included as part the ModusToolbox™ software.

Note: Many of these documents are provided online at the [ModusToolbox™ website](#). Also, some of the documents online might be more current than versions installed on disk.

Getting started documents

This section contains general documents to install and use ModusToolbox™ software, as well as use the provided Eclipse IDE.

ModusToolbox™ installation guide	This document describes how to install the ModusToolbox™ software on Windows, Linux, and macOS.
ModusToolbox™ tools package release notes	This document lists and describes features for this release of ModusToolbox™. It also includes known issues and workarounds and important design impacts you should know.
ModusToolbox™ user guide	This document provides an overall user guide for ModusToolbox™ GUI and CLI tools, including getting started and exporting to various IDEs, including Visual Studio Code, IAR Embedded Workbench, and Keil µVision.
Training material on GitHub	This is a comprehensive collection of information and exercises to help you learn how to use ModusToolbox™ software. It uses the CY8CKIT-062-43012 kit to demonstrate a variety of applications and features including MCU peripherals, FreeRTOS, Wi-Fi, Bluetooth®, and low power.
Eclipse IDE quick start guide	This is a short step-by-step guide specifically for using the Eclipse-based IDE to create and build applications.
Eclipse IDE user guide	This guide also focuses on the Eclipse IDE, covering more details about the IDE and software features.
Eclipse survival guide	This document is also online only. It offers tips on using the Eclipse environment.
EULA	End user license agreement; provided on disk as part of installation.

Configurator and tool documents

These documents are located in the "tools" directory in each individual configurator and tool "docs" subfolder.

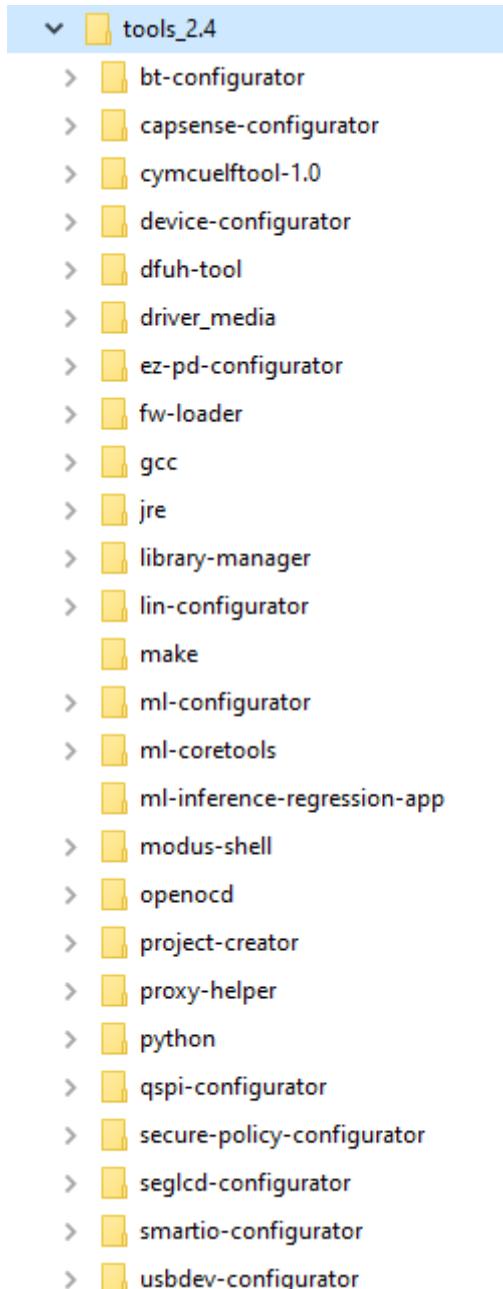
~~3 PSoC™ 6 software tools~~

~~3.3.1.3.3 IDE support~~

The ModusToolbox™ installer includes an optional Eclipse IDE that is a full-featured, cross-platform IDE. The ModusToolbox™ build system also provides support for Visual Studio (VS) Code, IAR Embedded Workbench, and Keil µVision. See the [Exporting to supported IDEs](#) chapter for more details.

~~3.3.1.3.4 Tools~~

The tools_2.4 directory includes the following configurators, tools, and utilities:



Configurators

Each configurator is a cross-platform tool that allows you to set configuration options for the corresponding hardware peripheral or library. When you save a configuration, the tool generates the C code and/or a

~~3 PSoC™ 6 software tools~~

configuration file used to initialize the hardware or library with the desired configuration. Configurators perform tasks such as:

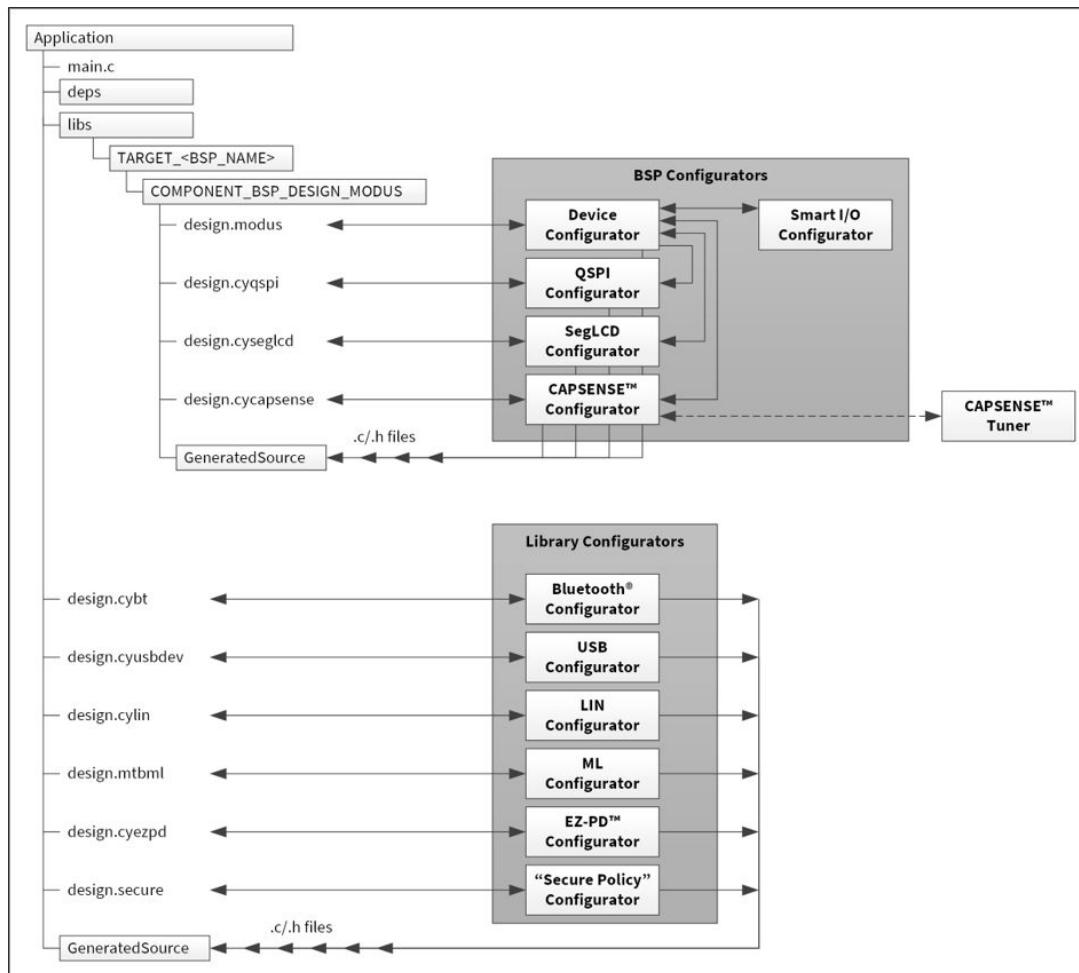
- Displaying a user interface for editing parameters
- Setting up connections such as pins and clocks for a peripheral
- Generating code to configure middleware

Configurators are 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 as part of a complete application. All of them are installed during the ModusToolbox™ installation. Each configurator provides a separate guide, available from the configurator's **Help** menu.

Note: Some configurators may not be useful for your application.

Configurators store configuration data in an XML data file that provides the desired configuration. Each configurator has a "command line" mode that can regenerate source based on the XML data file. Configurators are divided into two types: BSP Configurators and Library Configurators.

The following diagram shows a high-level view of the configurators that could be used in a typical application.



BSP configurators

BSP configurators configure the hardware on a specific device. This can be a board provided by us, a partner, or a board that you create that is specific to your application. Some of these configurators interact with the design.modus file to store and communicate configuration settings between different configurators. Code generated by a BSP Configurator is stored in a directory named GeneratedSource, which is in the same

~~3 PSoC™ 6 software tools~~

directory as the design.modus file. This is generally located in the BSP for a given target board. Some of the BSP configurators include:

- **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. Refer to the [Device Configurator guide](#) for more details.
- **CAPSENSE™ Configurator:** Configure CAPSENSE™ hardware, and generate the required firmware. This includes tasks such as mapping pins to sensors and how the sensors are scanned. Refer to the [CAPSENSE™ Configurator guide](#) for more details.

There is also a CAPSENSE™ Tuner to adjust performance and sensitivity of CAPSENSE™ widgets on the board connected to your computer. Refer to the [CAPSENSE™ Tuner guide](#) for more details.

- **QSPI Configurator:** Configure external memory and generate the required firmware. This includes defining and configuring what external memories are being communicated with. Refer to the [QSPI Configurator guide](#) for more details.
- **Smart I/O Configurator:** Configure the Smart I/O. This includes Chip, I/O, Data Unit, and LUT signals between port pins and the HSIOM. Refer to the [Smart I/O Configurator guide](#) for more details.
- **SegLCD Configurator:** Configure LCD displays. This configuration defines a matrix SegLCD connection and allows you to setup the connections and easily write to the display. Refer to the [SegLCD Configurator guide](#) for more details.

Library configurators

Library configurators support configuring application middleware. Library configurators do not read nor depend on the design.modus file. They generally create data structures to be consumed by software libraries. These data structures are specific to the software library and independent of the hardware. Configuration data is stored in a configurator-specific XML file (for example, *.cybt, *.cyusbdev, etc.). Any source code generated by the configurator is stored in a GeneratedSource directory in the same directory as the XML file. The Library configurators include:

- **Bluetooth® Configurator:** Configure Bluetooth® settings. These include options for specifying what services and profiles to use and what features to offer by creating SDP and/or GATT databases in generated code. This configurator supports both PSoC™ MCU and AIROC™ Bluetooth® applications. Refer to the [Bluetooth® Configurator guide](#) for more details.
- **USB Configurator:** Configure USB settings and generate the required firmware. This includes options for defining the Device Descriptor and Settings. Refer to the [USB Configurator guide](#) for more details.
- **LIN Configurator:** Configure various LIN settings, such as frames and signals, and generate the required firmware. Refer to the [LIN Configurator guide](#) for more details.
- **Machine Learning (ML) Configurator:** Accept a pre-trained ML model and generate an embedded model (as a library), which can be used along with your application code for a target device. Refer to the [ML Configurator guide](#) for more details.
- **EZ-PD™ Configurator:** Configure the features and parameters of the PDStack middleware for PMG1 family of devices. Refer to the [EZ-PD™ Configurator guide](#) for more details.
- **"Secure Policy" Configurator:** Open, create, and change policy configuration files for PSoC™ 64 "Secure Boot" MCU devices. Refer to the ["Secure Policy" Configurator guide](#) for more details.

Other tools

ModusToolbox™ software includes other tools that provide support for application creation, device firmware updates, and so on. All tools are installed by the [ModusToolbox™ tools package installer](#). With rare exception each tool has a user guide located in the docs directory beside the tool itself. Most user guides are also available online.

Other tools	Details	Documentation
project-creator	Create a new application. This tool is a stand-alone tool, available as a GUI and a command-line tool (CLI).	user guide

~~3 PSoC™ 6 software tools~~

Other tools	Details	Documentation
library manager	Add, remove, or update libraries and BSP used in an application; edits the Makefile	user guide
cymcuelftool	Merges CM0+ and CM4 application images into a single executable. Typically launched from a post-build script. This tool is not used by most applications.	user guide is in the tool's docs directory
dfuh-tool	Use the Device Firmware Update Host tool to communicate with a PSoC™ 6 MCU that has already been programmed with an application that includes device firmware update capability. Provided as a GUI and a command-line tool. Depending on the ecosystem you target, there may be other over-the-air firmware update tools available.	user guide

Utilities

ModusToolbox™ software includes some additional utilities that are often necessary for application development. In general, you use these utilities transparently.

Utility	Description
GCC	Supported toolchain included with the ModusToolbox™ installer.
GDB	The GNU Project Debugger is installed as part of GCC.
JRE	Java Runtime Environment; required by the Eclipse IDE integration layer.

Build system infrastructure

The build system infrastructure is the fundamental resource in ModusToolbox™ software. It serves three primary purposes:

- create an application, update and clone dependencies
- create an executable
- provide debug capabilities

A Makefile defines everything required for your application, including:

- target hardware (board/BSP to use)
- source code and libraries to use for the application
- ModusToolbox™ tools version, as well as compiler toolchain to use
- compiler/assembler/linker flags to control the build
- assorted variables to define things like file and directory locations

The build system automatically discovers all .c, .h, .cpp, .s, .a, .o files in the application directory and subdirectories, and uses them in the application. The Makefile can also discover files outside the application directory. You can add another directory using the CY_SHAREDLIB_PATH variable. You can also explicitly list files in the SOURCES and INCLUDES make variables.

Each library used in the application is identified by a .mtb file. This file contains the URL to a git repository, a commit tag, and a variable for where to put the library on disk. For example, a capsense.mtb file might contain the following line:

```
http://github.com/cypresssemiconductorco/capsense#latest-v2.X##$ASSET_REPO$$/capsense/latest-v2.X
```

~~3 PSoC™ 6 software tools~~

The build system implements the `make getlibs` command. This command finds each .mtb file, clones the specified repository, checks out the specified commit, and collects all the files into the specified directory. Typically, the `make getlibs` command is invoked transparently when you create an application or use the Library Manager, although you can invoke the command directly from a command line interface. See [ModusToolbox™ build system](#) for detailed documentation on the build system infrastructure.

Program and debug support

ModusToolbox™ software supports the [Open On-Chip Debugger](#) (OpenOCD) using a GDB server, and supports the J-Link debug probe. For the Mbed OS ecosystem, ModusToolbox™ supports Arm Mbed DAPLink.

You can use various IDEs to program devices and establish a debug session (see [Exporting to supported IDEs](#)). For programming, [CYPRESS™ Programmer](#) is available separately. It is a cross-platform application for programming PSoC™ 6 devices. It can program, erase, verify, and read the flash of the target device.

Cypress Programmer and the Eclipse IDE use KitProg3 low-level communication firmware. The firmware loader (fw-loader) is a software tool you can use to update KitProg3 firmware, if you need to do so. The fw-loader tool is installed with the ModusToolbox™ software. The latest version of the tool is also available separately in a [GitHub repository](#).

Tool	Description	Documentation
CYPRESS™ Programmer	CYPRESS™ Programmer functionality is built into ModusToolbox™ Software. It is also available as a stand-alone tool.	Programming tools page, go to the documentation tab
fw-loader	A simple command line tool to identify which version of KitProg is on a kit, and easily switch back and forth between legacy KitProg2 and current KitProg3.	readme.txt file in the tool directory
KitProg3	This tool is managed by fw-loader, it is not available separately. KitProg3 is a low-level communication/debug firmware that supports CMSIS-DAP and DAPLink (for Mbed OS). Use fw-loader to upgrade your kit to KitProg3, if needed.	user guide
OpenOCD	Our specific implementation of OpenOCD is installed with ModusToolbox™ software.	developer's guide
DAPLink	Support is implemented through KitProg3	DAPLink handbook

3.3.1.4 Product versioning

ModusToolbox™ products include tools and firmware that can be used individually, or as a group, to develop connected applications for our devices. We understand that you want to pick and choose the ModusToolbox™ products you use, merge them into your own flows, and develop applications in ways we cannot predict. However, it is important to understand that every tool and library may have more than one version. The tools package that provides the set of tools also has its own version. This section describes how ModusToolbox™ products are versioned.

3.3.1.4.1 General philosophy

ModusToolbox™ software is not a monolithic entity. Libraries and tools in the context of ModusToolbox™ are effectively "mini-products" with their own release schedules, upstream dependencies, and downstream dependent assets and applications. We deliver libraries via GitHub, and we deliver tools through the ModusToolbox™ installation package.

~~3 PSoC™ 6 software tools~~

All ModusToolbox™ products developed by us follow the standard versioning scheme:

- If there are known backward compatibility breaks, the major version is incremented.
- Minor version changes may introduce new features and functionality, but are "drop-in" compatible.
- Patch version changes address minor defects. They are very low-risk (fix the essential defect without unnecessary complexity).

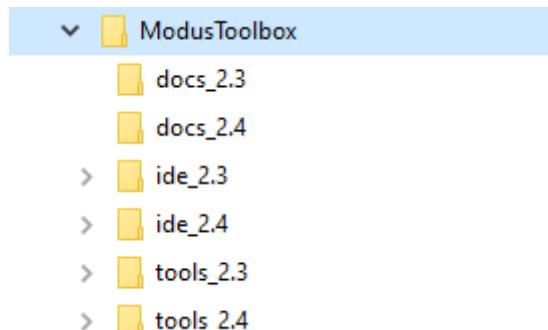
Code Examples include various libraries automatically. Prior to the ModusToolbox™ 2.3 release, these libraries were typically the latest versions. From the 2.3 release and newer, when you create a new application from a code example, any of the included libraries specified with a "latest-style" tag are converted to the "release-vX.Y.Z" style tag.

If you use the Library Manager to add a library to your project, the tool automatically finds and adds any required dependent libraries. From the 2.3 release and newer using the MTB flow, these dependencies are created using "release-vX.Y.Z" style tags. The tool also creates and updates a file named locking_commit.log in the deps subdirectory inside your application directory. This file maintains a history of all latest to release conversions made to ensure consistency with any libraries added in the future.

3.3.1.4.2 Tools package versioning

The ModusToolbox™ tools installation package is versioned as MAJOR.MINOR.PATCH. The file located at <install_path>/ModusToolbox/tools_2.4/version-2.4.0.xml also indicates the build number.

Every MAJOR.MINOR version of a ModusToolbox™ product is installed by default into <install_path>/ModusToolbox. So, if you have multiple versions of ModusToolbox™ software installed, they are all installed in parallel in the same ModusToolbox directory, as follows:



3.3.1.4.3 Multiple tools versions installed

When you run make commands from the command line, a message displays if you have multiple versions of the "tools" directory installed and if you have not specified a version to use.

~~3 PSoC™ 6 software tools~~

```

ckf@ORELPHOMF ~/examples_2.3>Hello_World
$ make help
Tools Directory: C:/Users/CKF/ModusToolbox/tools_2.3
CY8CKIT-062-WIFI-BT.mk: ..../mtb_shared/TARGET_CY8CKIT-062-WIFI-BT/latest-v2.X/CY8CKIT-062-WIFI-BT.mk

INFO: Multiple tools versions were found in CY_TOOLS_PATHS="/cygdrive/c/Users/CKF/ModusToolbox/tools_2.2 /cygdrive/c/Users/CKF/ModusToolbox/tools_2.3 C:/Users/CKF/ModusToolbox/tools_2.2 C:/Users/CKF/ModusToolbox/tools_2.3". This build is currently using CY_TOOLS_DIR="C:/Users/CKF/ModusToolbox/tools_2.3". Check that this is the correct version that should be used in this build. To stop seeing this message, explicitly set the CY_TOOLS_PATHS environment variable to the location of the tools directory. This can be done either as an environment variable or set in the application Makefile.

=====
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.
=====
```

3.3.1.4.4 Specifying alternate tools version

By default, the ModusToolbox™ software uses the most current version of the tools directory installed. That is, if you have ModusToolbox™ versions 2.3 and 2.4 installed, and if you launch the Eclipse IDE from the ModusToolbox™ 2.3 installation, the IDE will use the tools from the "tools_2.4" directory to launch configurators and build an application. This section describes how to specify the path to the desired version.

Environment variable

We support custom installation path.

For this and other reasons we support custom paths to be specific for the following, via corresponding system variables:

CY_TOOLS_PATHS

CY_GETLIBS_CACHE_PATH (there is also CY_GETLIBS_NO_CACHE)

CY_GETLIBS_OFFLINE_PATH (offline content)

CyManifestLocOverride (path to manifest.loc file)

In 3.0 we introducing the GLOBAL path for the assets like device-db (UDD).

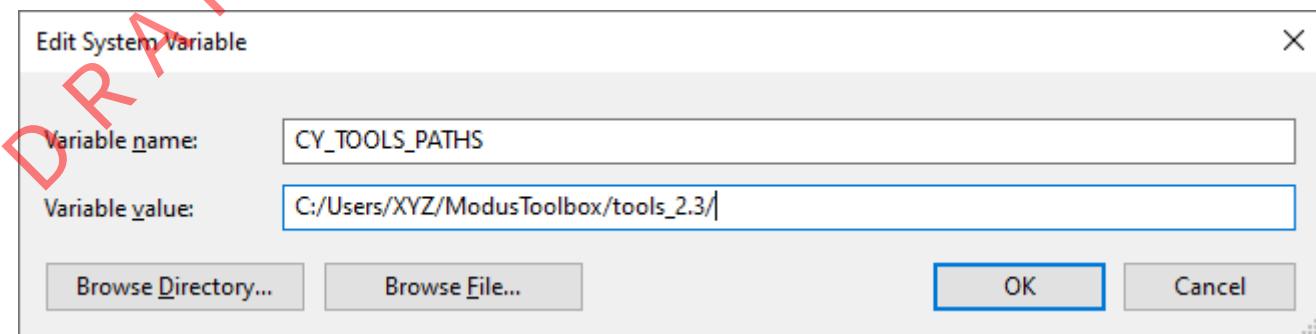
The environment variable that a user can set it CY_GETLIBS_GLOBAL_PATH. This gets normalized in core make and passed to the MTBQueryAPIs as MTB_GLOBAL_DIR. This is documented in the SAS.

The MTBQueryAPIs look for MTB_GLOBAL_DIR in the "make get_app_info" information. If this does not exist, it looks for the CY_GETLIBS_GLOBAL_PATH environment variable. Finally, if this does not exist, it assumes a default path of ~/.modustoolbox/global.

This is all implemented and in the MTBQueryAPIs.

The overall way to specify a path other than the default "tools" directory, is to use a system variable named CY_TOOLS_PATHS. On Windows, open the Environment Variables dialog, and create a new System/User Variable:

3 PSoC™ 6 software tools



Note: Use a Windows style path, (that is, not like /cygdrive/c/). Also, use forward slashes. For example:C:/Users/XYZ/ModusToolbox/tools_2.3/

Use the appropriate method for setting variables in macOS and Linux for your system.

Specific project Makefile

To preserve a specific "tools" path for the specific project, edit that project's Makefile, as follows:

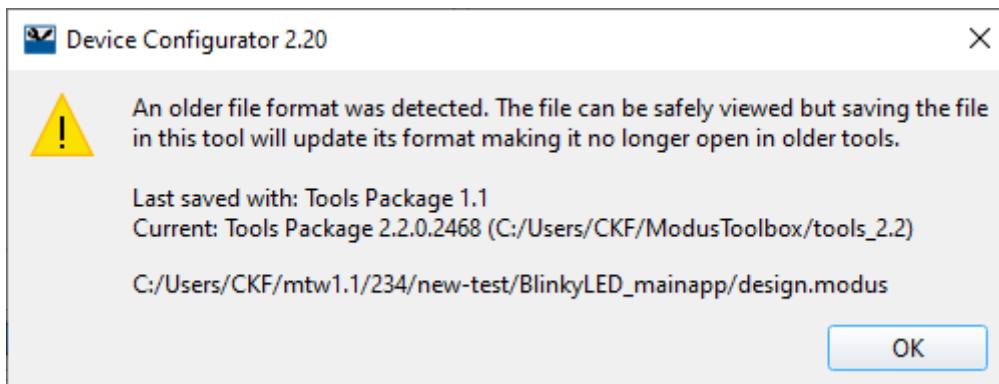
```
# If you install the IDE in a custom location, add the path to its
# "tools_X.Y" folder (where X and Y are the version number of the tools
# folder).
CY_TOOLS_PATHS+=C:/Users/XYZ/ModusToolbox/tools_2.3
```

3.3.1.4.5 Tools and configurators versioning

Every tool and configurator follow the standard versioning scheme and include a version.xml file that also contains a build number.

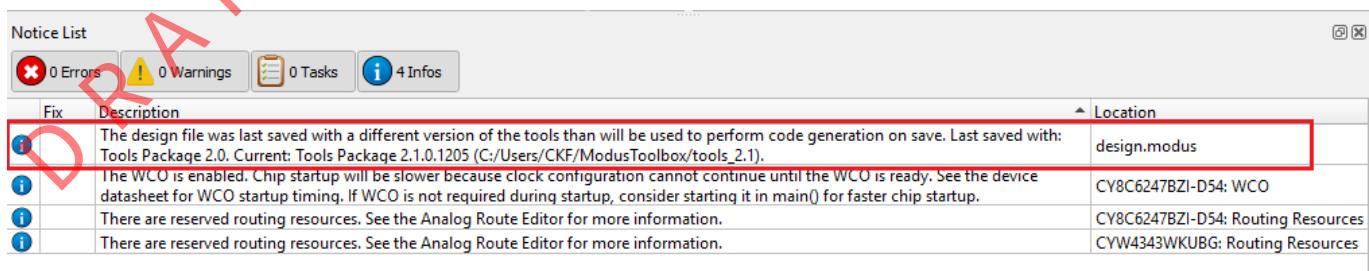
Configurator messages

Configurators indicate if you are about to modify the configuration file (for example, design.modus) with a newer version of the configurator, as well as if there is a risk that you will no longer be able to open it with the previous version of the configurator:



Configurators will also indicate if you are trying to open the existing configuration with a different, backward and forward compatible version of the Configurator.

3 PSoC™ 6 software tools



Fix	Description	Location
!	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 2.1.0.1205 (C:/Users/CKF/ModusToolbox/tools_2.1).	design.modus
!	The WCO is enabled. Chip startup will be slower because clock configuration cannot continue until the WCO is ready. See the device datasheet for WCO startup timing. If WCO is not required during startup, consider starting it in main() for faster chip startup.	CY8C6247BZI-D54: WCO
!	There are reserved routing resources. See the Analog Route Editor for more information.	CY8C6247BZI-D54: Routing Resources
!	There are reserved routing resources. See the Analog Route Editor for more information.	CYW4343WKUBG: Routing Resources

Note: *If using the command line, the build system will notify you with the same message.*

3.3.1.4.6 GitHub libraries versioning

GitHub libraries follow the same versioning scheme: MAJOR.MINOR.PATCH. The GitHub libraries, besides the code itself, also provide two files in MD format: README and RELEASE. The latter includes the version and the change history.

The versioning for GitHub libraries is implemented using GitHub tags. These tags are captured in the manifest files (see the [Manifest files](#) chapter for more details). The Project Creator tool parses the manifests to determine which BSPs and applications are available to select. The Library Manager tool parses the manifests and allow you to see and select between various tags of these libraries. When selecting a particular library of a particular version, the .mtb file gets created in your project. These .mtb files are a link to the specific tag. Refer to the [Library Manager user guide](#) for more details about tags.

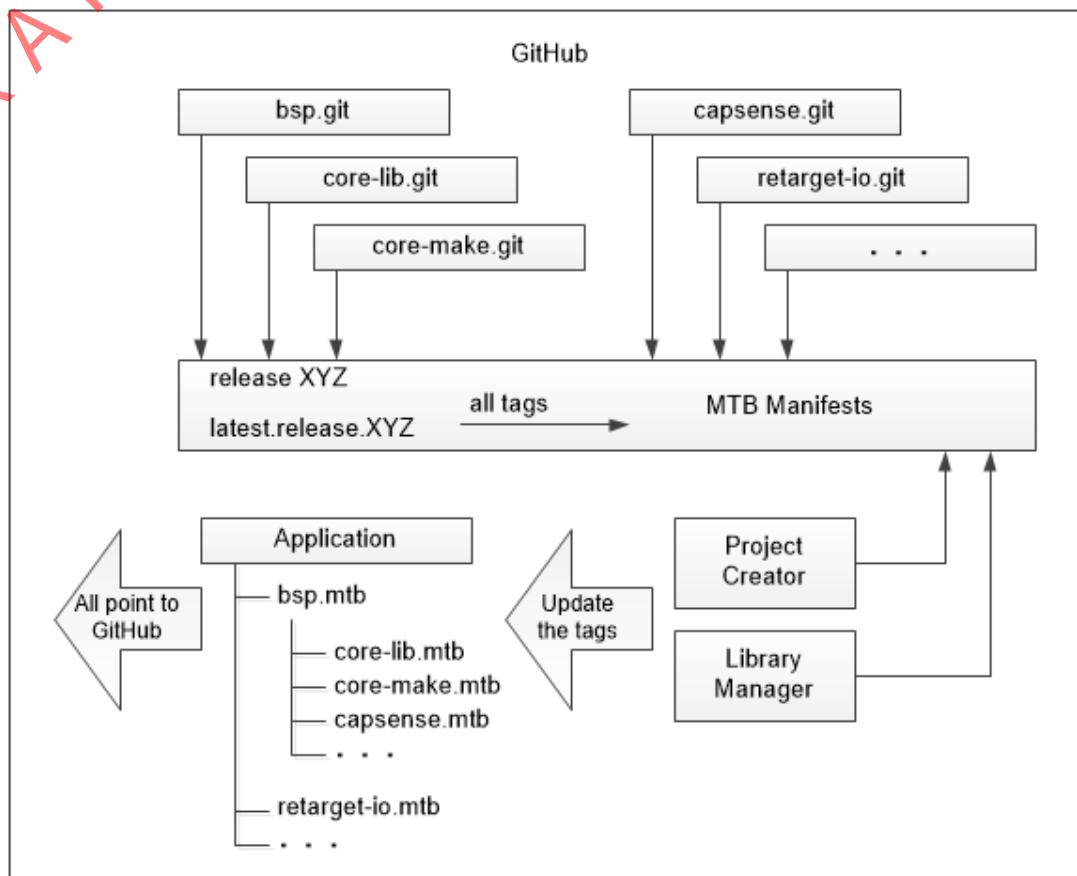
Once complete with initial development for your project, if using the git clone method to create the application instead of the Project Creator tool, we recommend you switch to specific "release" tags. Otherwise, running the make getlibs command will update the libraries referenced by the .mtb files, and will deliver the latest code changes for the major version.

3.3.1.4.7 Dependencies between libraries

The following diagram shows the dependencies between libraries.

3 PSoC™ 6 software tools

DRAFT



There are dependencies between the libraries. There are two types of dependencies:

Git repo dependencies via .mtb files

Dependencies for various libraries are specified in the manifest file. Only the top-level application will have .mtb files for the libraries it directly includes.

Regular C dependencies via #include

Our libraries only call the documented public interface of other Libraries. Every library declares its version in the header. The consumer of the library including the header checks if the version is supported, and will notify via #error if the newer version is required. Examples of the dependencies:

- The Device Support library (PDL) driver is used by the Middleware.
- The configuration generated by the Configurator depends on the versions of the device support library (PDL) or on the Middleware headers.

Similarly, if the configuration generated by the configurator of the newer version than you have installed, the notification via the build system will trigger asking you to install the newer version of the ModusToolbox™ software, which has a fragmented distribution model. You are allowed and empowered to update libraries individually.

3.3.1.5 Partner ecosystems

To support Infineon microcontrollers in our partner ecosystems, some tools and middleware from ModusToolbox™ software are also integrated into Mbed OS and Amazon FreeRTOS. Refer to [mbed.com](#) and [aws.amazon.com/freertos](#), respectively, to learn more about developing applications in those environments.

~~3 PSoC™ 6 software tools~~

~~3.3.2 Getting started~~

ModusToolbox™ software provides various graphical user interface (GUI) and command-line interface (CLI) tools to create and configure applications the way you want. You can use the included Eclipse-based IDE, which provides an integrated flow with all the ModusToolbox™ tools. Or, you can use other IDEs or no IDE at all. Plus, you can switch between GUI and CLI tools in various ways to fit your design flow. Regardless of what tools you use, the basic flow for working with ModusToolbox™ applications includes these tasks:

- [Install and configure software](#)
- [Get help](#)
- [Create applications](#)
- [Update BSPs and libraries](#)
- [Configure settings for devices, peripherals, and libraries](#)
- [Write application code](#)
- [Build, program, and debug](#)

This chapter helps you get started using various ModusToolbox™ tools. It covers these tasks, showing both the GUI and CLI options available.

3.3.2.1 Install and configure software

The ModusToolbox™ tools package is located on our website:

You can install the software on Windows®, Linux, and macOS. Refer to the [ModusToolbox™ installation guide](#) for specific instructions.

3.3.2.1.1 GUI set-up instructions

In general, the IDE and other GUI-based tools included as part of the ModusToolbox™ tools package work out of the box without any changes required. Simply launch the executable for the applicable GUI tool. On Windows, most tools are on the **Start** menu.

3.3.2.1.2 CLI set-up instructions

Before using the CLI tools, ensure that the environment is set up correctly.

- For Windows, the tools package provides a command-line utility called "modus-shell." You can run this from the **Start** menu, or navigate to the following installation directory and run *Cygwin.bat*:
`<install_path>/ModusToolbox/tools_2.4/modus-shell/`
- For macOS, the installer will detect if you have the necessary tools. If not, it will prompt you to install them using the appropriate Apple system tools.
- For Linux, there is only a ZIP file, and you are expected to understand how to set up various tools for your chosen operating system.

To check your installation, open the appropriate command-line shell.

- Type `which make`. For most environments, it should return `/usr/bin/make`.
- Type `which git`. For most environments, it should return `/usr/bin/git`.

If these commands return the appropriate paths, then you can begin using the CLI. Otherwise, install and configure the GNU make and git packages as appropriate for your environment.

3.3.2.2 Get help

In addition to this user guide, we provide documentation for both GUI and CLI tools. GUI tool documentation is generally available from the tool's **Help** menu. CLI documentation is available using the tool's `-h` option.

3 PSoC™ 6 software tools**3.3.2.2.1 GUI Documentation****Eclipse IDE**

If you choose to use the integrated Eclipse IDE, see the [Eclipse IDE for ModusToolbox™ quick start guide](#) for getting started information, and the [Eclipse IDE for ModusToolbox™ user guide](#) for additional details.

Configurator and tool guides

Each GUI-based configurator and tool includes a user guide that describes different elements of the tool, as well as how to use them. See [Development tools](#) for descriptions of these tools and links to the documentation.

3.3.2.2 Command line documentation**make help**

The ModusToolbox™ build system includes a make help target that provides help documentation. In order to use the help, you must first run the `make getlibs` command in an application directory (see [make getlibs](#) for details). From the appropriate shell in an application directory, type in the following to print the available make targets and variables to the console:

```
make help
```

To view verbose documentation for any of these targets or variables, specify them using the `CY_HELP` variable. For example:

```
make help CY_HELP=TOOLCHAIN
```

Note: *This help documentation is part of the base library, and it may also contain additional information specific to a BSP.*

To see the various make targets and variables available, see the [Available make targets](#) and [Available make variables](#) sections in the [ModusToolbox™ build system](#) chapter.

CLI tools

Various CLI tools include a `-h` option that prints help information to the screen about that tool. For example, running this command prints output for the Project Creator CLI tool to the screen:

```
./project-creator-cli -h
```

3 PSoC™ 6 software tools

~~CFE~~

```
ckf@ORELP8JPO0MF ~/ModusToolbox/tools_2.3/project-creator
$ ./project-creator-cli -h
Usage: C:\Users\CKF\ModusToolbox\tools_2.3\project-creator\project-creator-cli.exe [options]
The Project Creator is a stand-alone tool provided with the ModusToolbox software. It runs a "git clone" command to clone template
applications from a remote server onto your computer. The minimum required arguments of this tool are --board-id and --app-id.
The following example will clone a Hello World application configured for CY8CKIT-062-WIFI-BT BSP into your home directory:
  project-creator-cli --board-id CY8CKIT-062-WIFI-BT --app-id mtb-example-psoc6-hello-world
The options --list-boards and --list-apps show available BSPs and template applications, respectively.
Use the --target-dir option to specify the directory in which to clone the application other than <user-home> default (optional).
Use the --user-app-name option to specify the name of the application other than the template's default name (optional).

Options:
  -?, -h, --help           Displays this help.
  -v, --version            Displays version information.
  -a, --app-id <ID>        ID of the template application to clone.
  -b, --board-id <ID>      ID of the BSP to target.
  --list-apps <Board ID>   Lists IDs of all available template applications
                           for the given BSP.
  --list-boards             Lists IDs of all BSPs for which template
                           applications are available.
  --offline                Run in offline mode.
  -d, --target-dir <DIR>   (Optional) Target directory in which to create
                           the project. Default: <user-home>
  --use-modus-shell         If set, this tool uses binaries in
                           modus-shell/bin, like git, make instead of ones in
                           your PATH environment.
  -n, --user-app-name <NAME> (Optional) User-defined application name.
                           Default: Template app name.

ckf@ORELP8JPO0MF ~/ModusToolbox/tools_2.3/project-creator
$ -
```

3.3.2.3 Create applications

ModusToolbox™ software provides the Project Creator as both a GUI tool and a command line tool to easily create one or more ModusToolbox™ applications. See [Project Creator tools](#). If you prefer not to use the Project Creator tools, you can use the git clone command directly. See [git clone](#). However, be sure to also run the make getlibs command in the application directory. See [make getlibs](#). You can then use those application files in your preferred IDE or from the command line.

Note: Beginning with the ModusToolbox™ 2.2 release, we structure applications with the MTB flow. Using this flow, applications can share BSPs and libraries. If needed, different applications can use different versions of the same BSP/library. Sharing resources reduces the number of files on your computer and speeds up subsequent application creation time. Shared BSPs, libraries, and versions are located in the 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. Refer to the [Library Manager User Guide](#) for details.

Looking ahead, most example applications will use the 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™ software fully supports both flows, but it only supports one flow or the other for a given application.

For simplicity, this guide focuses on the MTB flow. For details about how the LIB flow works, refer to the ModusToolbox™ 2.1 revision of this guide, located here:

3.3.2.3.1 Project Creator tools

The Project Creator tools run the git clone command for the selected code example(s) and create a directory at the specified location with the specified name. The tools also updates the application Makefile and create a <BSP-NAME>.mtb file based on the specified BSP. That .mtb file contains the following:

- The URL of the git repo where the BSP contents can be found.
- The commit (version of the library) to checkout / make visible / use in the application.
- A variable of where to put the BSP on disk (shared or local to the application).

The Project Creator tools then run the make getlibs command to read the BSP manifest file, resolve dependencies, and import libraries. Depending on the settings in the application and manifest, the tools put

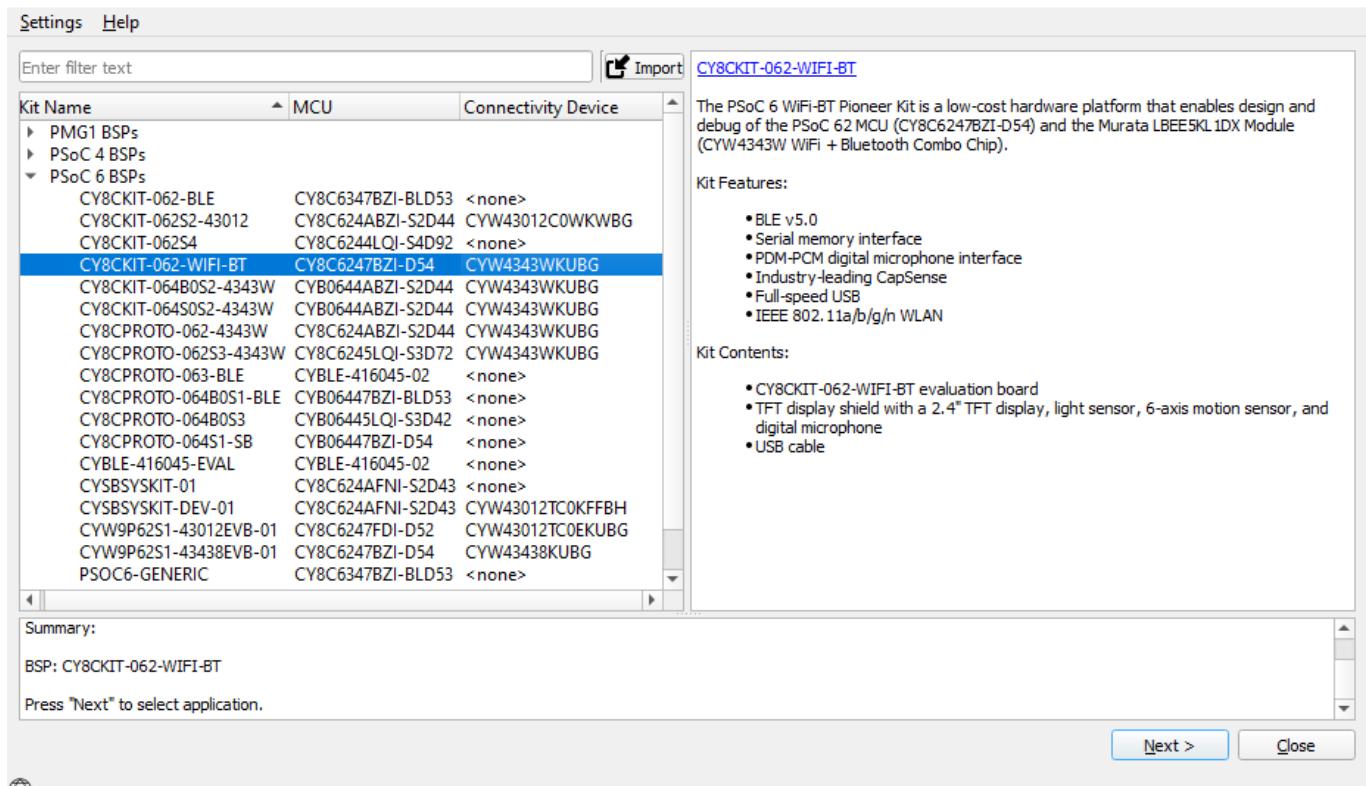
~~3 PSoC™ 6 software tools~~

everything into application directories and an mtb_shared directory. In most cases, BSPs are placed local to the application, while libraries are shared.

Project Creator GUI

The Project Creator GUI tool provides a series of screens to select a BSP and code example, specify the application name and location, as well as select target IDE. The tool displays various messages during the application creation process. Refer to the [Project Creator user guide](#) for more details. Open the Project Creator GUI tool from the Windows Start menu or by running the executable file installed in the following directory by default:

<install_path>/ModusToolbox/tools_2.4/project-creator/



The option to select a target IDE generates necessary files for that IDE. If you launch the Project Creator GUI tool from the included Eclipse-based IDE, it seamlessly exports the created application for use in the Eclipse IDE.

project-creator-cli

You can also use the project-creator-cli tool to create applications from a command-line prompt or from within batch files or shell scripts. The tool is located in the same directory as the GUI version (<install_path>/ModusToolbox/tools_2.4/project-creator/). To see all the options available, run the tool with the -h option:

```
./project-creator-cli -h
```

~~3 PSoC™ 6 software tools~~

The following example shows running the tool with various options.

```
./project-creator-cli \
--board-id CY8CKIT-062-WIFI-BT \
--app-id mtb-example-psoc6-hello-world \
--user-app-name MyLED \
--target-dir "C:/my_projects"
```

In this example, the project-creator-cli tool runs the git clone command to clone the Hello World code example from our GitHub server (<https://github.com/Infineon>). It also updates the TARGET variable in the Makefile to match the selected BSP (--board-id), creates a .mtb file for it, and runs the make getlibs command to obtain the necessary library files. This example also includes options to specify the name (--user-app-name) and location (--target-dir) where the application will be stored.

3.3.2.3.2 git clone

The Project Creator GUI and command line tools run the git clone command as part of the process of creating an application. You can run the git clone command directly from the command line. Open the appropriate shell and type in the following command (replace the <URL> with the appropriate URL of the repo you wish to clone):

```
git clone <URL>
```

The clone operation creates an application directory in your current location. Navigate to that directory (`cd <DIR>`), and find the application Makefile. This is the top-level file that determines the application build flow. To see the various make targets and variables that you can edit in this file, refer to the [Available make targets](#) and [Available make variables](#) sections in the [ModusToolbox™ build system](#) chapter.

Note: *When using the git clone command directly, be sure to also run the make getlibs command in the application directory. See [make getlibs](#). Also, each code example has a default BSP included in the application's deps subdirectory. If you want to use a different BSP, you must create a .mtb file for it in the deps subdirectory before running make getlibs, and you must change the TARGET variable in the Makefile.*

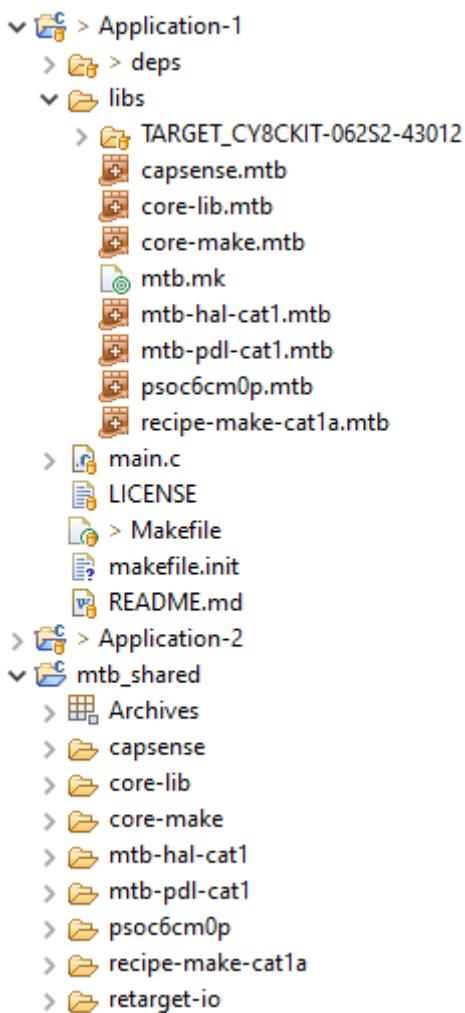
Note: *The git clone command does not automatically lock the libraries to the latest versions. Therefore, when you use make getlibs in future updates, your libraries may be updated to newer versions. You can use Library Manager to manually lock the library versions.*

3.3.2.3.3 Typical application contents

After an application has been created for the MTB flow and all the libraries have been imported, it contains the following basic files and directories as shown in the following image:

3 PSoC™ 6 software tools

DRAFT



Application directory

This directory contains the application source code, Makefile, readme file, as well as the deps and libs subdirectories. If you create multiple applications, there will be multiple application directories contained in the same directory structure or workspace.

- Source code – This is one or more files for your application's code. Often it is named main.c, but it could be more than one file and the files could have almost any name. Source code files can also be grouped into a subdirectory anywhere in the application's directory (for example, sources/main.c).
- Makefile – This is the application's Makefile, which contains configuration information. See the [ModusToolbox™ build system](#) chapter for more details.
- deps subdirectory – By default, this subdirectory contains .mtb files using the MTB flow.
 - Initially, this subdirectory contains only the <BSP>.mtb file for the BSP you selected for the application.
 - It could also contain <library>.mtb files for libraries that were included directly or for which you changed using the Library Manager. See the [Update BSPs and libraries](#) section for details.
 - This subdirectory also contains the locking_commit.log file, which keeps track of the version for each dependent library.
- libs subdirectory – This subdirectory may contain different types of files generated by the [make getlibs](#) process, based on how the application is created. You can regenerate these files using the [make getlibs](#) command, so you do not need to add these files to source control.
 - This subdirectory contains BSPs that are local to the application (that is, not shared).

~~3 PSoC™ 6 software tools~~

- If you update your application to specify any libraries to be local as well, then this directory will also contain source code for those libraries.
- By default, this subdirectory contains the <library>.mtb files for libraries included as indirect dependencies of the BSP or other libraries.
- This directory also contains the mtb.mk file that lists the shared libraries and their versions.

mtb_shared directory

Typically, a new application also includes a mtb_shared directory adjacent to the application directory, and this is where the shared BSP and libraries are cloned by default. This location can be modified by specifying the CY_GETLIBS_PATH variable. Duplicate libraries are checked to see if they point to the same commit and if so, only one copy is kept in the mtb_shared directory. You can regenerate these files using the make getlibs command, so you do not need to add these files to source control.

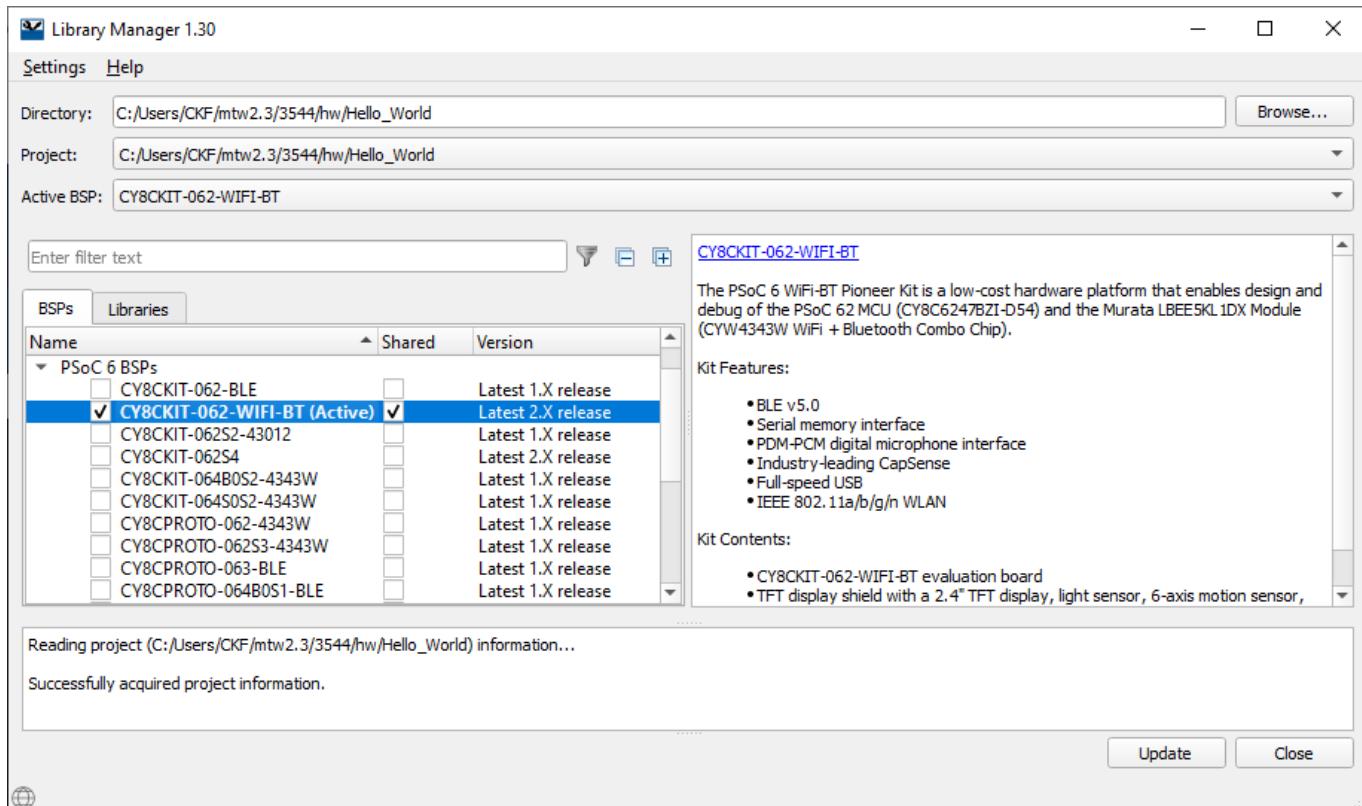
3.3.2.4 Update BSPs and libraries

As part of the application creation process, the Project Creator tools update the application with BSP and library information. If you use the git clone command, you will have to update BSP and library information as a separate process using the Library Manager tool or from the command line using the make getlibs command. You can also update the BSP and library information at any point in the development cycle using these tools.

3.3.2.4.1 Library Manager

As needed, use the Library Manager tool to add or remove BSPs and libraries for your application, as well as change versions for BSPs and libraries. You can also change the active BSP. Open the Library Manager tool from the application directory using the make modlibs command.

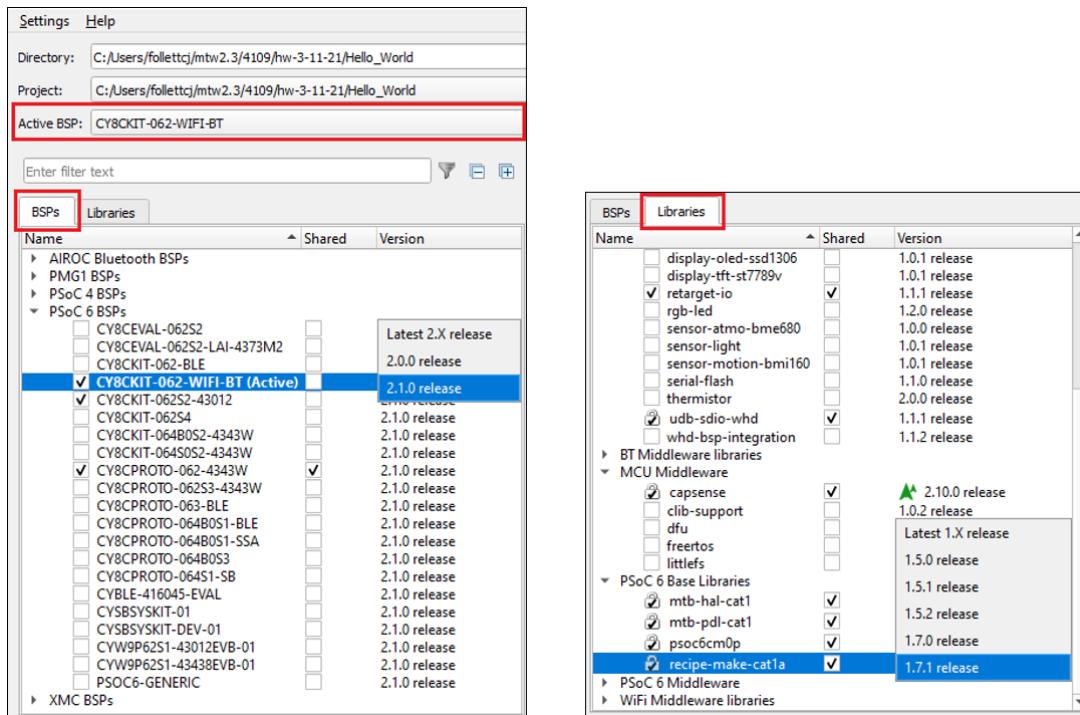
The Library Manager opens for the selected application and its available BSPs and libraries.



~~3 PSoC™ 6 software tools~~

Note: There are several ways to open the Library Manager; refer to the [Library Manager user guide](#) for more details.

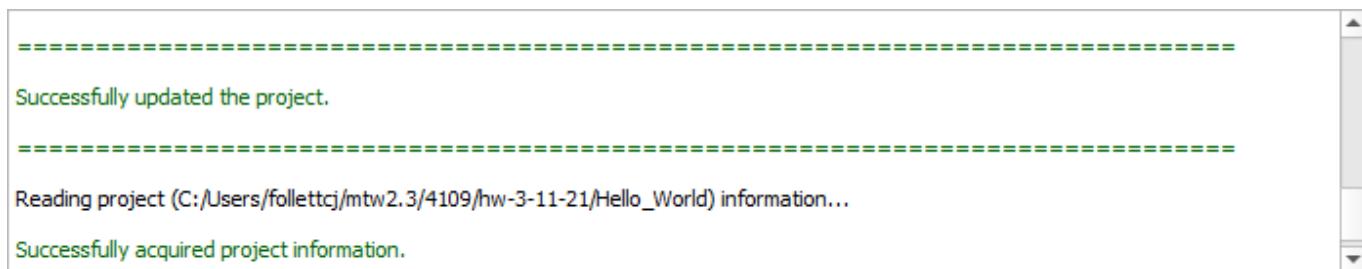
The Library Manager tool provides a field to select the Active BSP. It also includes two tabs to view and update BSPs and Libraries.



Make changes to BSPs and libraries as follows:

- Select one or more check boxes under **Name** for the items to add. Deselect check boxes for items to remove.
- Specify whether items are shared (placed in the mtb_shared directory) or local to the application (placed in the libs subdirectory) by selecting/deselecting the **Shared** check box.
- Choose an appropriate Version for each item.

Click **Update** to proceed with the changes. The status box displays various messages while applying changes, and then indicates if the application was updated or not.



3.3.2.4.2 make getlibs

In the MTB flow, the Project Creator tools and the Library Manager tool run the `make getlibs` command to search for all .mtb files in the application directory. Each .mtb file contains information used when the application is created. These files are parsed, and the libraries are cloned into a directory named `mtb_shared`.

If you ran the `git clone` command manually and did not use the Library Manager, then your application will contain only default .mtb files. You must run the `make getlibs` command to parse those files and clone the libraries. However, if you want to use a different BSP than the default provided by the code example, you

~~3 PSoC™ 6 software tools~~

must first edit the Makefile to update the TARGET variable to match the desired BSP. Then, you must add a .mtb file in the /deps subdirectory that includes a URL to the desired BSP location.

Note: ModusToolbox™ applications that use the LIB flow contain.lib files in the deps subdirectory. If an application uses the MTB flow, then all .lib files are ignored.

When you are ready to update your application, open the appropriate shell (see [CLI set-up instructions](#)) and run the following command in the application directory:

```
make getlibs
```

Note: The `make getlibs` operation may take a long time to execute as it depends on your internet speed and the size of the libraries that it is cloning. To improve subsequent library cloning operations, a cache directory named ".modustoolbox/cache" exists in the \$HOME (Linux, macOS) and \$USERPROFILE (Windows) directories.

3.3.2.5 Configure settings for devices, peripherals, and libraries

Depending on your application, you may want to update and generate some of the configuration code. While it is possible to write configuration code from scratch, the effort to do so is considerable. ModusToolbox™ software provides applications called configurators that make it easier to configure a hardware block or a middleware library. 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.

Before configuring your device, you must decide how your application will interact with the hardware; see [Application layers](#). That decision affects how you configure settings for devices, peripherals, and libraries.

Note: Before you make changes to settings in configurators, you should first copy the configuration information to the application and override the BSP configuration or create a custom BSP. See details about BSPs in the [Board support packages chapter](#). If you make changes to a standard BSP library, it will cause the repo to become dirty. Additionally, if the BSP is in the shared asset repository, changes will impact all applications that use the shared BSP. If this happens, refer to [KBA231252](#).

The configurators can be run as GUIs to easily update various parameters and settings. Most can also be run as command line tools to regenerate code as part of a script. For more information about configurators, see the [Configurators](#) section. Also, each configurator provides a separate document, available from the configurator's [Help](#) menu, that provides information about how to use the specific configurator.

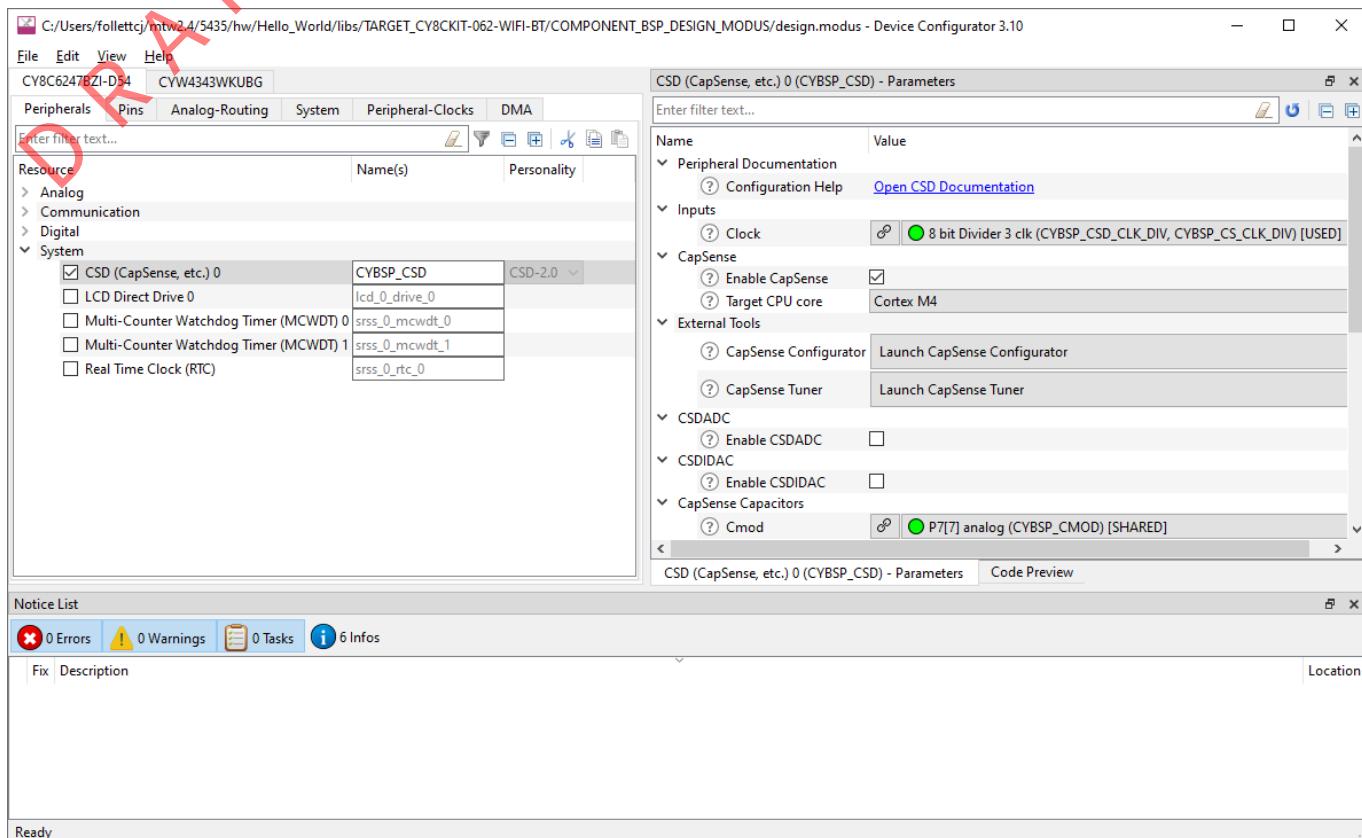
3.3.2.5.1 Configurator GUI tools

You can open various configurator GUIs using the appropriate make command from the application directory. For example, to open the Device Configurator, run:

```
make config
```

This opens the Device Configurator with the current application's design.modus configuration file.

3 PSoC™ 6 software tools



As described under [Tools make targets](#), you can use the make open command with appropriate arguments to open any configurator. For example, to open the CAPSENSE™ Configurator, run:

```
make open CY_OPEN_TYPE=capsense-configurator
```

You can also use the Eclipse IDE provided with ModusToolbox™ software to open configurators. For example, if you select the "Device Configurator" link in the IDE Quick Panel, the tool opens with the application's design.modus file. Refer to the [Eclipse IDE for ModusToolbox™ user guide](#) for more details about the Eclipse IDE. One other way to open BSP configurators (such as CAPSENSE™ and SegLCD Configurators) is by using a link from inside the Device Configurator. However, this does not apply to Library configurators (such as Bluetooth® and USB Configurators).

3.3.2.5.2 Configurator CLI tools

Most of the configurators can also be run from the command line. The primary use case is to re-generate source code based on the latest configuration settings. This would often be part of an overall build script for the entire application. The command-line configurator cannot change configuration settings. For information about command line options, run the configurator using the -h option.

3.3.2.6 Write application code

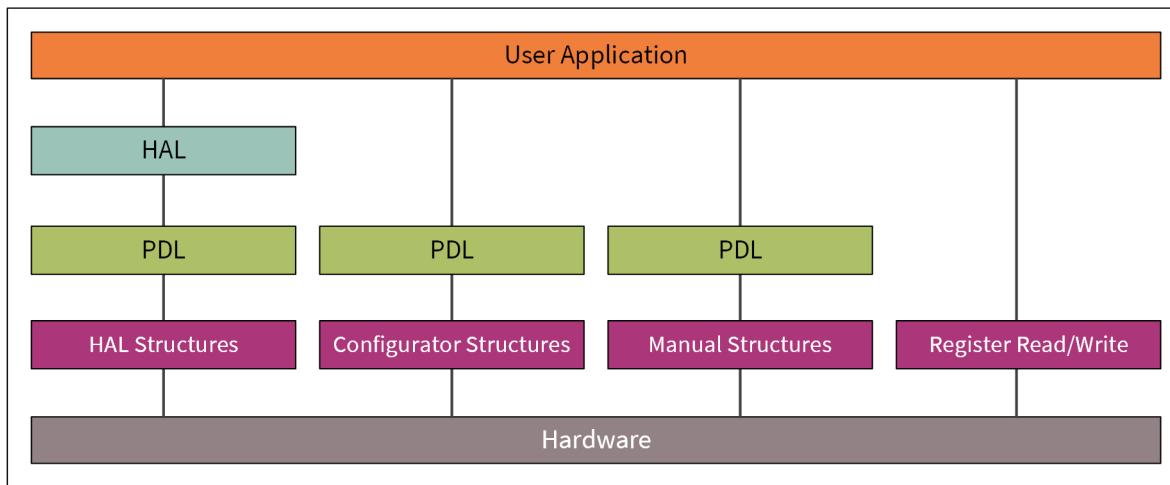
As in any embedded development application using any set of tools, you are responsible for the design and implementation of the firmware. This includes not just low-level configuration and power mode transitions, but all the unique functionality of your product. When writing application code, you must decide how the application will interact with the hardware; see [Application layers](#).

~~3 PSoC™ 6 software tools~~

ModusToolbox™ software enables your workflow. It includes an integrated Eclipse IDE, as well as support for Visual Studio (VS) Code, IAR Embedded Workbench, and Keil µVision (see [Exporting to supported IDEs](#)). You can also use a text editor and command line tools. Taken together, the multiple resources available to you in ModusToolbox™ software: BSPs, configurators, driver libraries, and middleware, help you focus on your specific application.

3.3.2.6.1 Application layers

There are four distinct ways for an application to interact with the hardware as shown in the following diagram:



- HAL structures: Application code uses the HAL, which interacts with the PDL through structures created by the HAL
- Configurator structures: Application code uses PDL through structures created by a Configurator.
- Manual structures: Application code uses PDL through structures created manually.
- Register read/write: Application code uses direct register read and writes.

Note: A single application may use different methods for different peripherals.

HAL

Using the HAL is more portable than the other methods. It is the preferred method for simpler functions and those that don't have extremely strict flash size limitations. It is a high-level interface to the hardware that allows many common functions to be done quickly and easily. This allows the same code to be used even if there are changes to pin assignments, different devices in the same family, or even to a different family that may have radically different underlying architectures. For more details, refer to [HAL on GitHub](#).

The advantages include:

- Easy hardware changes. Just change the pin assignment in the BSP and the code remains the same. For example, if LED1 changes from P0_0 to P0_1, the code remains the same as long as the code uses the name LED1 with the HAL. The only change is to the BSP pin assignment.
- Easy migration to a different device as product requirements change.
- Ability to use the same code base across multiple projects and generations, even if underlying architectures are different.

The disadvantages include:

- The HAL may not support every feature that the hardware has. It supports the most common features but not all of them to maintain simplicity.
- The HAL will use additional flash space. The additional flash depends on which HAL APIs are used.

~~3 PSoC™ 6 software tools~~

~~PDL~~

The PDL is a lower-level interface to the hardware (but still simpler than direct register access) that supports all hardware features. Usually the PDL goes hand-in-hand with Configurators, which will be described next. Since the PDL interacts with the hardware at a lower level it is less portable between devices, especially those with different architectures. For more details, refer to [PDL on GitHub](#).

The advantages/disadvantages are the exact opposite of those for the HAL. The main advantage is that it provides access to every hardware feature.

Configurators

Configurators make initial setup easier for hardware accessed using the PDL. The Configurators create structures that the PDL requires without you needing to know the exact composition of each structure, and they create the proper structure based on your selections. See [Configurators](#) for more information.

If you use the HAL for a peripheral, it will create the necessary structures for you, so you should NOT use a Configurator to set them up. The HAL structure is accessible, and once you initialize a peripheral with the HAL you can view and even modify that structure (that is, a HAL object). The underlying structures are hardware-specific, so you may be sacrificing portability if you modify the structure manually. There are a few exceptions. For example, it is reasonable to configure system items (such as clocks) and use them with the HAL.

3.3.2.7 Build, program, and debug

After the application has been created, you can export it to an IDE of your choice for building, programming, and debugging. You can also use command line tools. The ModusToolbox™ build system infrastructure provides several make variables to control the build. So, whether you are using an IDE or command line tools, you edit the Makefile variables as appropriate. See the [ModusToolbox™ build system](#) chapter for detailed documentation on the build system infrastructure.

Variable	Description
TARGET	Specifies the target board/kit. For example, CY8CPROTO-062-4343W
APPNAME	Specifies the name of the application
TOOLCHAIN	Specifies the build tools used to build the application
CONFIG	Specifies the configuration option for the build [Debug Release]
VERBOSE	Specifies whether the build is silent or verbose [true false]

ModusToolbox™ software is tested with various versions of the TOOLCHAIN values listed in the following table. Refer to the release information for each product for specific versions of the toolchains.

TOOLCHAIN	Tools	Host OS
GCC_ARM	GNU Arm Embedded Compiler	Mac OS, Windows, Linux
ARM	Arm compiler	Windows, Linux
IAR	Embedded Workbench	Windows

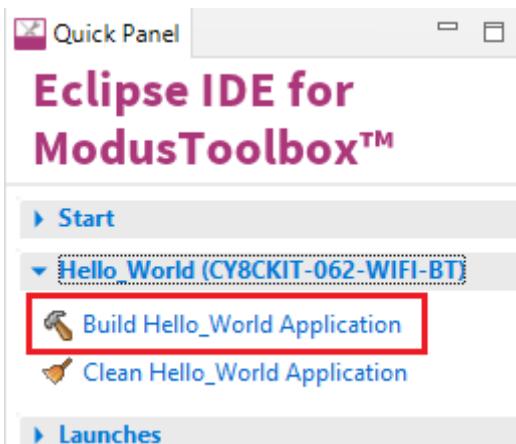
In the Makefile, set the TOOLCHAIN variable to the build tools of your choice. For example: TOOLCHAIN=GCC_ARM. There are also variables you can use to pass compiler and linker flags to the toolchain.

ModusToolbox™ software installs the GNU Arm toolchain and uses it by default. If you wish to use another toolchain, you must provide it and specify the path to the tools. For example, `CY_COMPILER_PATH=<yourpath>`. If this path is blank, the build infrastructure looks in the ModusToolbox/ install directory.

~~3 PSoC™ 6 software tools~~

~~3.3.2.7.1~~ Use Eclipse IDE

When using the provided Eclipse IDE, click the Build Application link in the Quick Panel for the selected application.



Because the IDE relies on the build infrastructure, it does not use the standard Eclipse GUI to modify build settings. It uses the build options specified in the Makefile. This design ensures that the behavior of the application, its options, and the make process are consistent regardless of the development environment and workflow.

If you do change settings in the Makefile (for example, TARGET or CONFIG), you must re-create the launch configs using the link in the Quick Panel; refer to the [Eclipse IDE for ModusToolbox™ user guide](#) for more details.

3.3.2.7.2 Export to another IDE

If you prefer to use an IDE other than Eclipse, you can select the appropriate IDE from the Target IDE pull-down menu when creating an application using the Project Creator tool. You can also use the appropriate "make <ide>" command. For example, to export to Visual Studio Code, run:

```
make vscode
```

For more details about using other IDEs, see the [Exporting to supported IDEs](#) chapter. When working with a different IDE, you must manage the build using the features and capabilities of that IDE.

3.3.2.7.3 Use command line

make build

When all the libraries are available (after executing `make getlibs`), the application is ready to build. From the appropriate shell, type the following:

```
make build
```

3 PSoC™ 6 software tools

This instructs the build system to find and gather the source files in the application and initiate the build process. In order to improve the build speed, you may "parallelize" it by giving it a -j flag (optionally specifying the number of processes to run). For example:

```
make build -j16
```

make program

Connect the target board to the machine and type the following in the shell:

```
make program
```

This performs an application build and then programs the application artifact (usually an .elf or .hex file) to the board using the recipe-specific programming routine (usually OpenOCD). You may also skip the build step by using `make qprogram` instead of `make program`. This will program the existing build artifact.

make debug/qdebug/attach

The following commands can be used to debug the application, as follows:

- `make debug` – Build and program the board. Then launch the GDB server.
- `make qdebug` – Skip the build and program steps. Just launch the GDB server.
- `make attach` – Starts a GDB client and attaches the debugger to the running target.

~~3 PSoC™ 6 software tools~~

~~3.3.3 ModusToolbox™ build system~~

This chapter covers various aspects of the ModusToolbox™ build system. Refer to [CLI set-up instructions](#) for getting started information about using the command line tools. This chapter is organized as follows:

- [Overview](#)
- [Application types](#)
- [BSPs](#)
- [make getlibs](#)
- [Adding source files](#)
- [Pre-builds and post-builds](#)
- [Program and debug](#)
- [Available make targets](#)
- [Available make variables](#)

3.3.3.1 Overview

The ModusToolbox™ build system is based on GNU make. It performs application builds and provides the logic required to launch tools and run utilities. It consists of a light and accessible set of Makefiles deployed as part of every application. This structure allows each application to own the build process, and it allows environment-specific or application-specific changes to be made with relative ease. The system runs on any environment that has the make and git utilities. For a "how to" document about the ModusToolbox™ Makefile system, refer to <https://community.cypress.com/docs/DOC-18994>. Also, as described in the [Getting started](#) chapter, you can run the make help command to get details on the various targets and variables available.

The ModusToolbox™ command line interface (CLI) and supported IDEs all use the same build system. Hence, switching between them is fully supported. Program/Debug and other tools can be used in either the command line or an IDE environment. In all cases, the build system relies on the presence of ModusToolbox™ tools included with the ModusToolbox™ installer.

The tools contain a start.mk file that serves as a reference point for setting up the environment before executing the recipe-specific build in the base library. The file also provides a getlibs make target that brings libraries into an application. Every application must then specify a target board on which the application will run. These are provided by the <BSP>.mk files deployed as a part of a BSP library.

The majority of the Makefiles are deployed as git repositories (called "repos"), in the same way that libraries are deployed in the ModusToolbox™ software. There are two separate repos: core-make used by all recipes and a recipe-make-xxx that contains BSP/target specific details. These are the minimum required to enable an application build. Together, these Makefiles form the build system.

3.3.3.2 Application types

The build system supports the following application types:

- Normal application – The application consists of one application Makefile. The build process creates one artifact. All pre-built libraries are brought in at link time. A normal application is constructed by defining the APPNAME variable in the application Makefile.
- Library application – The application consists of one application Makefile. The sources are built into a library. These libraries may be linked in as part of a Normal application build. A library application is constructed by defining the LIBNAME variable in the application Makefile.

~~3 PSoC™ 6 software tools~~

The library applications are usually placed as companions to normal applications. These normal applications specify their dependency on library applications by including them in the DEPENDENT_LIB_PATHS make variable. They also drive the build process of the library applications by defining a shared_libs make target. For example:

```
DEPENDENT_LIB_PATHS=../bspLib
shared_libs:
    make -C ../bspLib build -j
```

3.3.3.3 BSPs

An application must specify a target BSP through the TARGET variable in the Makefile. We provide reference BSPs for its development kits. Use these as a reference to construct your own BSP. For more information about BSPs, refer to the [Board support packages](#) chapter.

- When using the Project Creator to create an application, it provides the selected BSP and updates the Makefile.
- Use the Library Manager to add, update, or remove a BSP from an application. You can also add a .mtb file that contains the URL and a version tag of interest in the application.

3.3.3.4 make getlibs

When you run the `make getlibs` command, the build system finds all the .mtb files in the application directory and performs `git clone` operations on them. A .mtb file contains a git URL to a library repo, a specific tag for a version of the code, and a variable to specify the location to store the library.

The `getlibs` target finds and processes all .mtb files and uses the `git` command to clone or pull the code as appropriate. The target also calls the `library-manager-cli` tool to generate .mtb files for indirect dependencies. Then, it checks out the specific tag listed in the .mtb file. The Project Creator and Library Manager invoke this process automatically.

The `make getlibs` target:

- Must be invoked separately from any other make target (for example, the command `make getlibs build` is not allowed and the Makefiles will generate an error; however, a command such as `make clean build` is allowed).
- Performs a `git fetch` on existing libraries, but will always checkout the tag pointed to by the overseeing .mtb file.
- Detects if users have modified standard code and will not overwrite their work. This allows you to perform some action (for example commit code or revert changes, as appropriate) instead of overwriting the changes.

The build system also has a `printlibs` target that can be used to print the status of the cloned libraries.

3.3.3.4.1 repos

The cloned libraries are located in their individual git repos in the directory pointed to by the CY_GETLIBS_PATH variable (for example, `/deps`). These all point to the "our" remote origin. You can point your repo by editing the `.git/config` file or by running the `git remote` command.

If the repos are modified, add the changes to your source control (`git branch` is recommended). When `make getlibs` is run (to either add new libraries or update libraries), it requires the repos to be clean. You may also use the `.gitignore` file for adding untracked files when running `make getlibs`. See also [KBA231252](#).

~~3 PSoC™ 6 software tools~~

~~3.3.3.5 Adding source files~~

Source and header files placed in the application directory hierarchy are automatically added by the auto-discovery mechanism. Similarly, library archives and object files are automatically added to the application. Any object file not referenced by the application is discarded by the linker. The Project Creator and Library Manager tools run the `make_getlibs` command and generate a `mtb.mk` file in the application's `libs` subdirectory. This file specifies the location of shared libraries included in the build.

The application Makefile can also include specific source files (`SOURCES`), header file locations (`INCLUDES`) and pre-built libraries (`LDLIBS`). This is useful when the files are located outside of the application directory hierarchy or when specific sources need to be included from the filtered directories.

3.3.3.5.1 Auto-discovery

The build system implements auto-discovery of library files, source files, header files, object files, and pre-built libraries. If these files follow the specified rules, they are guaranteed to be brought into the application build automatically. Auto-discovery searches for all supported file types in the application directory hierarchy and performs filtering based on a directory naming convention and specified directories, as well as files to ignore. If files external to the application directory hierarchy need to be added, they can be specified using the `SOURCES`, `INCLUDES`, and `LIBS` make variables.

Auto-discovery of source code (source and headers) has no depth limit (it uses the "find" tool). Auto-discovery of other types of files do have a depth limit, including:

- .mtb file depth
- .mk file of the selected TARGET
- device support library discovery
- configurator file discovery

The default depth limit for these files is five directories deep. They can be changed to up to nine directories deep by setting the following options in the Makefile:

```
CY_UTILS_SEARCH_DEPTH=9
CY_LIBS_SEARCH_DEPTH=9
```

To control which files are included/excluded, the build system implements a filtering mechanism based on directory names and `.cyignore` files.

.cyignore

Prior to applying auto-discovery and filtering, the build system will first search for `.cyignore` files and construct a set of directories and files to exclude. It contains a set of directories and files to exclude, relative to the location of the `.cyignore` file. The `.cyignore` file can contain make variables. For example, you can use the `SEARCH_` variable to exclude code from other libraries as shown for the "Test" directory in a library called `<library-name>`:

```
$(SEARCH_<library-name>)/Test
```

The `CY_IGNORE` variable can also be used in the Makefile to define directories and files to exclude.

Note: The `CY_IGNORE` variable should contain paths that are relative to the application root. For example, `./src1`.

~~3 PSoC™ 6 software tools~~

~~TOOLCHAIN_<NAME>~~

Any directory that has the prefix "TOOLCHAIN_" is interpreted as a directory that is toolchain specific. The "NAME" corresponds to the value stored in the TOOLCHAIN make variable. For example, an IAR-specific set of files is located under a directory named TOOLCHAIN_IAR. Auto-discovery only includes the TOOLCHAIN_<NAME> directories for the specified TOOLCHAIN. All others are ignored. ModusToolbox™ software supports IAR, ARM, and GCC_ARM.

~~TARGET_<NAME>~~

Any directory that has the prefix "TARGET_" is interpreted as a directory that is target specific. The "NAME" corresponds to the value stored in the TARGET make variable. For example, a build with TARGET=CY8CPROTO-062-4343W ignores all TARGET_ directories except TARGET_CY8CPROTO-062-4343W.

Note: *The TARGET_ directory is often associated with the BSP, but it can be used in a generic sense. For example, if application sources need to be included only for a certain TARGET, this mechanism can be used to achieve that.*

Note: *The output directory structure includes the TARGET name in the path, so you can build for target A and B and both artifact files will exist on disk.*

~~CONFIG_<NAME>~~

Any directory that has the prefix "CONFIG_" is interpreted as a directory that is configuration (Debug/Release) specific. The "NAME" corresponds to the value stored in the CONFIG make variable. For example, a build with CONFIG=CustomBuild ignores all CONFIG_ directories, except CONFIG_CustomBuild.

Note: *The output directory structure includes the CONFIG name in the path, so you can build for config A and B and both artifact files will exist on disk.*

~~COMPONENT_<NAME>~~

Any directory that has the prefix "COMPONENT_" is interpreted as a directory that is component specific. The "NAME" corresponds to the value stored in the COMPONENT make variable. For example, consider an application that sets COMPONENTS+=comp1. Also assume that there are two directories containing component-specific sources:

```
COMPONENT_comp1/src.c
COMPONENT_comp2/src.c
```

Auto-discovery will only include COMPONENT_comp1/src.c and ignore COMPONENT_comp2/src.c. If a specific component needs to be removed, either delete it from the COMPONENTS variable or add it to the DISABLE_COMPONENTS variable.

BSP makefile

Auto-discovery will also search for a <TARGET>.mk file (aka, BSP makefile). If no matching BSP makefile is found, it will fail to build. This file can also be manually specified by setting it in the CY_EXTRA_INCLUDES variable.

~~DO NOT USE~~ 3 PSoC™ 6 software tools

Multi-project application with imported BSP

When you use an imported BSP to create a multi-project application, the system copies the BSP into an application root folder. For these types of applications, the Project Creator tool creates an importedbsp.mk file for each project with a SEARCH variable and relative path to the imported BSP. For example:

```
SEARCH+=<relative_path_to_BSP_folder>
```

If you do not use the Project Creator tool, you must create the files manually in each project directory.

In addition, when `make getlibs` is run, it updates the mtb.mk file with the following line:

```
-include ${CY_INTERNAL_APP_PATH}/importedbsp.mk
```

The "-" in front of "include" tells the make system to perform a conditional include. It only includes the file if it exists. If the file does not exist, the system does not issue a warning.

3.3.3.6 Pre-builds and post-builds

A pre-build or post-build operation is typically a script file invoked by the build system. Such operations are possible at several stages in the build process. They can be specified at the application, BSP, and recipe levels. You can pre-build and post-build arguments in the application Makefile. For example:

```
PREBUILD=command -arg1 -arg2
```

If you want to run more than one command, separate them with a semicolon (;). For example:

```
PREBUILD=command1 -arg1; command2 -arg1 -arg2
```

The sequence of execution in a build is as follows:

1. BSP pre-build – Defined using CY_BSP_PREBUILD variable.
2. Application pre-build – Defined using PREBUILD variable.
3. Source generation – Defined using CY_RECIPE_GENSRC variable.
4. Recipe pre-build – Defined using CY_RECIPE_PREBUILD variable.
5. Source compilation and linking.
6. Recipe post-build – Defined using CY_RECIPE_POSTBUILD variable.
7. BSP post-build – Defined using CY_BSP_POSTBUILD variable.
8. Application post-build – Defined using POSTBUILD variable.

The variable value is the relative path to the script to be executed.

Note: *Pre-builds happen after the auto-discovery process. Therefore, if the pre-build steps generate any source files to be included in a build, they will not be automatically included until the subsequent build. In this scenario, this step should use the \$(shell) function directly in the application Makefile instead of using the provided pre-build make variables. For example: \$(shell bash ./custom_gen.sh ARG1 ARG2)*

~~3 PSoC™ 6 software tools~~

~~3.3.3.7 Program and debug~~

The programming step can be done through the CLI by using the following make targets:

- `program` – Build and program the board.
- `qprogram` – Skip the build step and program the board.
- `debug` – Build and program the board. Then launch the GDB server.
- `qdebug` – Skip the build and program steps. Just launch the GDB server.
- `attach` – Starts a GDB client and attaches the debugger to the running target.

For CLI debugging, the `attach` target must be run on a separate shell instance. Use the GDB commands to debug the application.

3.3.3.8 Available make targets

A make target specifies the type of function or activity that the make invocation executes. The build system does not support a make command with multiple targets. Therefore, a target must be called in a separate make invocation. The following tables list and describe the available make targets for all recipes.

3.3.3.8.1 General make targets

Target	Description
<code>all</code>	Same as build. That is, builds the application. This target is equivalent to the build target.
<code>getlibs</code>	Clones the repositories and checks out the identified commit. The repos are cloned to the <code>libs</code> directory. By default, this directory is created in the application directory. It may be directed to other locations using the <code>CY_GETLIBS_PATH</code> variable.
<code>build</code>	Builds the application. The build process involves source auto-discovery, code-generation, pre-builds, and post-builds. For faster incremental builds, use the <code>qbuild</code> target to skip the auto-discovery step.
<code>qbuild</code>	Quick builds the application using the previous build's source list. When no other sources need to be auto-discovered, this target can be used to skip the auto-discovery step for a faster incremental build.
<code>program</code>	Builds the artifact and programs it to the target device. The build process performs the same operations as the build target. Upon successful completion, the artifact is programmed to the board.
<code>qprogram</code>	Quick programs a built application to the target device without rebuilding. This target allows programming an existing artifact to the board without a build step.
<code>debug</code>	Builds and programs. Then launches a GDB server. Once the GDB server is launched, another shell should be opened to launch a GDB client.
<code>qdebug</code>	Skip the build and program step and does quick debug; that is, it launches a GDB server. Once the GDB server is launched, another shell should be opened to launch a GDB client.
<code>attach</code>	Starts a GDB client and attaches the debugger to the running target.
<code>clean</code>	Cleans the <code>/build/<TARGET></code> directory. The directory and all its contents are deleted from disk.

~~3 PSoC™ 6 software tools~~

Target	Description
help	<p>Prints the help documentation.</p> <p>Use the CY_HELP=<name of target or variable> to see the verbose documentation for a given target or a variable.</p>

3.3.3.8.2 IDE make targets

Target	Description
eclipse	<p>Generates Eclipse IDE launch configurations and project files.</p> <p>This target expects the CY_IDE_PRJNAME variable to be set to the name of the application as defined in the Eclipse IDE. For example, make eclipse CY_IDE_PRJNAME=AppV1. If this variable is not defined, it will use the APPNAME for the launch configs. This target also generates .cproject and .project files if they do not exist in the application root directory.</p>
vscode	<p>Generates VS Code IDE json files.</p> <p>This target generates VS Code json files for debug/program launches, IntelliSense, and custom tasks. These overwrite the existing files in the application directory except for settings.json.</p>
ewarm8	<p>Generates IAR-EW version 8 IDE .ipcf file.</p> <p>This target requires you to also set TOOLCHAIN=IAR. It generates an IAR Embedded Workbench v8.x compatible .ipcf file that can be imported into IAR-EW. The .ipcf file is overwritten every time this target is run.</p>
uvision5	<p>Generates Keil µVision v5 IDE .cpdsc, .gpdsc, and .cprj files.</p> <p>This target requires you to also set TOOLCHAIN=ARM. It generates a CMSIS compatible .cpdsc and .gpdsc files that can be imported into Keil µVision v5. Both files are overwritten every time this target is run.</p>

3.3.3.8.3 Tools make targets

Target	Description
open	<p>Opens/launches a specified tool. This is intended for use by the Eclipse IDE. Use make config, make config_bt, or make config_usbdev instead.</p> <p>This target accepts two variables: CY_OPEN_TYPE and CY_OPEN_FILE. At least one of these must be provided. The tool can be specified by setting the CY_OPEN_TYPE variable. A specific file can also be passed using the CY_OPEN_FILE variable. If only CY_OPEN_FILE is given, the build system will launch the default tool associated with the file's extension.</p> <p>Supported types are: bt-configurator capsense-configurator capsense-tuner device-configurator dfuh-tool library-manager project-creator qspi-configurator seglcd-configurator smartio-configurator usbdev-configurator.</p>
modlibs	<p>Launches the library-manager executable for updating libraries.</p> <p>The Library Manager can be used to add/remove libraries and to upgrade/downgrade existing libraries.</p>
config	<p>Runs the Device Configurator on the target *.modus file.</p> <p>If no existing device-configuration files are found, the configurator is launched to create one.</p>
config_bt	<p>Runs the Bluetooth® Configurator on the target *.cybt file.</p> <p>If no existing bt-configuration files are found, the configurator is launched to create one.</p>

~~3 PSoC™ 6 software tools~~

Target	Description
config_usbdev	<p>Runs the USB Configurator on the target *.cyusbdev file.</p> <p>If no existing usbdev-configuration files are found, the configurator is launched to create one.</p>
config_secure	<p>Runs the "Secure Policy" Configurator.</p> <p>This configurator is intended only for devices that support secure provisioning.</p>
config_ezpd	<p>Runs the EZ-PD™ Configurator.</p> <p>If no existing ez-pd-configuration files are found, the configurator is launched to create one.</p>
config_lin	<p>Runs the LIN configurator.</p> <p>If no existing lin-configuration files are found, the configurator is launched to create one.</p>

3.3.3.8.4 Utility make targets

Target	Description
progtool	<p>Performs specified operations on the programmer/firmware-loader.</p> <p>This target expects user-interaction on the shell while running it. When prompted, you must specify the command(s) to run for the tool.</p>
bsp	<p>Generates a TARGET_GEN board/kit from TARGET.</p> <p>This target generates a new BSP with the name provided in TARGET_GEN based on the current TARGET. The TARGET_GEN variable must be populated with the name of the new TARGET. Optionally, you may define the target device (DEVICE_GEN) and additional devices (ADDITIONAL_DEVICES_GEN) such as radios. For example:</p> <pre>make bsp TARGET_GEN=NewBoard DEVICE_GEN=CY8C624ABZI-S2D44 ADDITIONAL_DEVICES_GEN=CYW4343WKUBG</pre>
update_bsp	<p>Change the device in a custom BSP generated by the make bsp command.</p> <p>This target changes the device set in a custom BSP generated by the make bsp command. The TARGET_GEN variable will specify the BSP to modify. The DEVICE_GEN variable will specify the new target device of the BSP. For example:</p> <pre>make update_bsp TARGET_GEN=NewBoard DEVICE_GEN=CY8C624ABZI-S2D44</pre>
lib2mtbx	<p>Convert .lib files to .mtbx files</p> <p>This will recursively look for .lib files in CONVERSION_PATH and create equivalent .mtbx files adjacent to them. The type of .mtbx file is determined by the CONVERSION_TYPE variable. This can be either [local] or [shared]. The default is [local].</p>

3 PSoC™ 6 software tools

DRAFT

Target	Description
import_deps	<p>Import dependent .mtbx files of a given path into the application.</p> <p>This will recursively look for .mtbx files in <code>IMPORT_PATH</code>, copy them to the application's deps directory and rename them to .mtb files. This makes them direct dependencies of the application. Note that the import process is not applicable for applications using .lib files. These libraries must instead be situated in the application directory. This process does not automatically lock the libraries to the latest version; use the Library Manager to lock versions.</p>
get_app_info	<p>Prints the application info for the Eclipse IDE for ModusToolbox™.</p> <p>The file types can be specified by setting the <code>CY_CONFIG_FILE_EXT</code> variable. For example:</p> <pre>make get_app_info CY_CONFIG_FILE_EXT="modus cybt cyusbdev"</pre>
get_env_info	<p>Prints the make, git, and application repo info.</p> <p>This allows a quick printout of the current application repo and the make and git tool locations and versions.</p>
printlibs	<p>Prints the status of the library repos.</p> <p>This target parses through the library repos and prints the SHA1 commit. It also shows whether the repo is clean (no changes) or dirty (modified or new files).</p>
check	<p>Checks for the necessary tools.</p> <p>Not all tools are necessary for every build recipe. This target allows you to get an idea of which tools are missing if a build fails in an unexpected way.</p>

3.3.3.9 Available make variables

The following variables customize various make targets. They can be defined in the application Makefile or passed through the make invocation. The following sections group the variables for how they can be used.

3.3.3.9.1 Basic configuration make variables

These variables define basic aspects of building an application. For example:

```
make build TOOLCHAIN=GCC_ARM CONFIG=CustomConfig -j8
```

Variable	Description
TARGET	<p>Specifies the target board/kit (that is, BSP). For example, CY8CPROTO-062-4343W.</p> <p>Example usage: <code>make build TARGET=CY8CPROTO-062-4343W</code></p>
APPNAME	<p>Specifies the name of the application. For example, "AppV1" > AppV1.elf.</p> <p>Example usage: <code>make build APPNAME="AppV1"</code></p> <p>This variable is used to set the name of the application artifact (programmable image). It also signifies that the application will build for a programmable image artifact that is intended for a target board. For applications that need to build to an archive (library), use the <code>LIBNAME</code> variable.</p>

~~3 PSoC™ 6 software tools~~

Variable	Description
LIBNAME	<p>Specifies the name of the library application. For example, "LibV1" > LibV1.a.</p> <p>Example Usage: <code>make build LIBNAME="LibV1"</code></p> <p>This variable is used to set the name of the application artifact (prebuilt library). It also signifies that the application will build an archive (library) that is intended to be linked to another application. These library applications can be added as dependencies to an artifact producing application using the <code>DEPENDENT_LIB_PATHS</code> variable.</p>
TOOLCHAIN	<p>Specifies the toolchain used to build the application. For example, <code>GCC_ARM</code>.</p> <p>Example Usage: <code>make build TOOLCHAIN=IAR CY_COMPILER_PATH=<path>/IAR Systems/Embedded Workbench 8.4/arm/</code></p> <p>Supported toolchains for this include <code>GCC_ARM</code>, <code>IAR</code>, and <code>ARM</code>.</p>
CONFIG	<p>Specifies the configuration option for the build [Debug Release].</p> <p>Example Usage: <code>make build CONFIG=Release</code></p> <p>The <code>CONFIG</code> variable is not limited to Debug/Release. It can be other values. However in those instances, the build system will not configure the optimization flags. <code>Debug=lowest optimization</code>, <code>Release=highest optimization</code>.</p> <p>The optimization flags are toolchain specific. If you go with your custom config, then you can manually set the optimization flag in the <code>CFLAGS</code>.</p>
VERBOSE	<p>Specifies whether the build is silent [false] or verbose [true].</p> <p>Example Usage: <code>make build VERBOSE=true</code></p> <p>Setting <code>VERBOSE</code> to true may help in debugging build errors/warnings. By default, it is set to false.</p>

3.3.3.9.2 Advanced configuration make variables

These variables define advanced aspects of building an application.

Variable	Description
SOURCES	<p>Specifies C/C++ and assembly files outside of application directory.</p> <p>Example Usage (within Makefile): <code>SOURCES+=path/to/file/Source1.c</code></p> <p>This can be used to include files external to the application directory. The path can be both absolute or relative to the application directory.</p>
INCLUDES	<p>Specifies include paths outside of the application directory.</p> <p>Example Usage (within Makefile): <code>INCLUDES+=path/to/headers</code></p>
DEFINES	<p>Specifies additional defines passed to the compiler.</p> <p>Example Usage (within Makefile): <code>DEFINES+=EXAMPLE_DEFINE=0x01</code></p>
VFP_SELECT	<p>Selects hard/soft ABI for floating-point operations [softfp hardfp]. If not defined, this value defaults to softfp.</p> <p>Example Usage (within Makefile): <code>vfp_select=hardfp</code></p>
CFLAGS	<p>Prepends additional C compiler flags.</p> <p>Example Usage (within Makefile): <code>cflags+= -Werror -Wall -O2</code></p>
CXXFLAGS	<p>Prepends additional C++ compiler flags.</p> <p>Example Usage (within Makefile): <code>cxxflags+= -finline-functions</code></p>
ASFLAGS	<p>Prepends additional assembler flags.</p>

~~3 PSoC™ 6 software tools~~

Variable	Description
LDFLAGS	Prepends additional linker flags. Example Usage (within Makefile): LDFLAGS+= -nodefaultlibs
LDLIBS	Includes application-specific prebuilt libraries. Example Usage (within Makefile): LDLIBS+=./MyBinaryFolder/binary.o
LINKER_SCRIPT	Specifies a custom linker script location. Example Usage (within Makefile): LINKER_SCRIPT=path/to/file/Custom_Link1.ld This linker script overrides the default.
PREBUILD	Specifies the location of a custom pre-build step and its arguments. This operation runs before the build recipe's pre-build step. Example Usage (within Makefile): PREBUILD=python example_script.py If the default pre-build step needs to be replaced, define the CY_RECIPES_PREBUILD make variable with the desired pre-build step.
POSTBUILD	Specifies the location of a custom post-build step and its arguments. This operation runs after the build recipe's post-build step. Example Usage (within Makefile): POSTBUILD=python example_script.py
COMPONENTS	Adds component-specific files to the build. Example Usage (within Makefile): COMPONENTS+=CUSTOM_CONFIGURATION Create a directory named COMPONENT_<VALUE> and place your files. Then provide <VALUE> to this make variable to have that feature library be included in the build. For example, create a directory named COMPONENT_ACCELEROMETER. Then include it in the make variable: COMPONENT=ACCELEROMETER. If the make variable does not include the <VALUE>, then that library will not be included in the build.
DISABLE_COMPONENTS	Removes component-specific files from the build. Example Usage (within Makefile): DISABLE_COMPONENTS=BSP_DESIGN_MODUS Include a <VALUE> to this make variable to have that feature library be excluded in the build. For example, to exclude the contents of the COMPONENT_BSP_DESIGN_MODUS directory, set DISABLE_COMPONENTS=BSP_DESIGN_MODUS as shown.
DEPENDENT_LIB_PATHS	List of dependent library application paths. For example, ../bspLib. An artifact-producing application (defined by setting APPNAME) can have a dependency on library applications (defined by setting LIBNAME). This variable defines those dependencies for the artifact-producing application. The actual build invocation of those libraries is handled at the application level by defining the shared_libs target. For example:
DEPENDENT_APP_PATHS	List of dependent application paths. For example, ../cy_m0p_image. The main application can have a dependency on other artifact producing applications (defined by setting APPNAME). This variable defines those dependencies for the main application. The artifacts of these dependent applications are translated to c-arrays and are brought into the main application as regular c-files and are compiled and linked. The main application also generates a "standalone" variant of the main application that does not include the dependent applications.

3 PSoC™ 6 software tools

Variable	Description
SEARCH	List of paths to include in auto-discovery. For example, .. /mtb_shared/lib1. When make getlibs is run for applications that use .mtb files, a file is generated in ./libs/mtb.mk. This file automatically populates the SEARCH variable with the locations of the libraries in the shared repo location (set by the CY_GETLIBS_SEARCH_PATH and CY_GETLIBS_SHARED_NAME variables). The SEARCH variable can also be used by the application to include other directories to auto-discovery.
IMPORT_PATH	Path to .mtbx dependency files to import into the application. This variable must be defined when calling make import_deps. Any .mtbx dependency file found in this directory will be imported into the application and will become a direct dependency.
CONVERSION_PATH	Path to the .lib files to convert to .mtbx files. This variable must be defined when calling make lib2mtbx. Any .lib file found in this directory will be converted.
CONVERSION_TYPE	(optional) Defines the type of .mtbx file to create. This variable can be set to [local] or [shared]. The default type is [local]. If [local], the library will be deposited in the application's CY_GETLIBS_PATH directory when performing make getlibs. If [shared], the library will be deposited (when running make getlibs) in the shared location as defined by CY_GETLIBS_SHARED_PATH and CY_GETLIBS_SHARED_NAME.
FORCE	Optional) Force overwrite existing files. When this variable is set [true], make lib2mtbx overwrites any existing .mtbx files.

3.3.3.9.3 BSP make variables

These variables are used with the make bsp target.

Variable	Description
DEVICE	Device ID for the primary MCU on the target board/kit (set by TARGET.mk). The device identifier is mandatory for all board/kits.
TARGET_GEN	Name of the new target board/kit. Example Usage: make bsp TARGET_GEN=MyBSP This is a mandatory variable when calling the bsp make target. It is used to name the board/kit files and directory.
DEVICE_GEN	(Optional) Device ID for the primary MCU on the new target board/kit. Example Usage: make bsp TARGET_GEN=MyBSP DEVICE_GEN=CY8C624ABZI-S2D44 This is an optional variable when calling the bsp make target to replace the primary MCU on the board (overwrites DEVICE). If it is not defined, the new board/kit will use the existing DEVICE from the board/kit that it is copying from.

3.3.3.9.4 Getlibs make variables

These variables are used with the make getlibs target.

~~3 PSoC™ 6 software tools~~

Variable	Description
CY_GETLIBS_NO_CACHE	<p>Disables the cache when running getlibs.</p> <p>Example Usage: <code>make getlibs CY_GETLIBS_NO_CACHE=true</code></p> <p>To improve the library creation time, the <code>make getlibs</code> target uses a cache located in the user's home directory (<code>\$HOME</code> for macOS/Linux and <code>\$USERPROFILE</code> for Windows). Disabling the cache allows 3rd-party libraries to be brought in to the application using .mtb files just like our libraries.</p>
CY_GETLIBS_OFFLINE	<p>Use the offline location as the library source.</p> <p>Example Usage: <code>make getlibs CY_GETLIBS_OFFLINE=true</code></p> <p>Setting this variable signals to the build system to use the offline location (Default: <code><HOME>/modustoolbox/offline</code>) when running the <code>make getlibs</code> target. The location of the offline content can be changed by defining the <code>CY_GETLIBS_OFFLINE_PATH</code> variable.</p>
CY_GETLIBS_PATH	<p>Absolute path to the intended location of libs directory.</p> <p>Example Usage: <code>make getlibs CY_GETLIBS_PATH="path/to/directory"</code></p> <p>The library repos are cloned into a directory named, <code>libs</code> (default: <code><CY_APP_PATH>/libs</code>). Setting this variable allows specifying the location of the <code>libs</code> directory to be elsewhere on disk.</p>
CY_GETLIBS_DEPS_PATH	<p>Absolute path to where the library-manager stores .mtb and .lib files. Usage is similar to <code>CY_GETLIBS_PATH</code>.</p> <p>Setting this path allows relocating the directory that the library-manager uses to store the .mtb / .lib files in your application. The default location is in a directory named <code>/deps</code> (Default: <code><CY_APP_PATH>/deps</code>).</p>
CY_GETLIBS_CACHE_PATH	<p>Absolute path to the cache directory. Usage is similar to <code>CY_GETLIBS_PATH</code>.</p> <p>The build system caches all cloned repos in a directory named <code>/cache</code> (Default: <code><HOME>/modustoolbox/cache</code>). Setting this variable allows the cache to be relocated to elsewhere on disk. To disable the cache entirely, set the <code>CY_GETLIBS_NO_CACHE</code> variable to [true].</p>
CY_GETLIBS_OFFLINE_PATH	<p>Absolute path to the offline content directory. Usage is similar to <code>CY_GETLIBS_PATH</code>.</p> <p>The offline content is used to create/update applications when not connected to the internet (Default: <code><HOME>/modustoolbox/offline</code>). Setting this variable allows to relocate the offline content to elsewhere on disk.</p>
CY_GETLIBS_SEARCH_PATH	<p>Relative path to the top directory for getlibs operation. Usage is similar to <code>CY_GETLIBS_PATH</code>.</p> <p>The <code>make getlibs</code> operation by default executes at the location of the <code>CY_APP_PATH</code>. This can be overridden by specifying this variable to point to a specific location.</p>
CY_GETLIBS_SHARED_PATH	<p>Relative path to the shared repo location.</p> <p>All .mtb files have the format, <code><URI><COMMIT><LOCATION></code>. If the <code><LOCATION></code> field begins with <code>\$\$ASSET_REPO\$\$</code>, then the repo is deposited in the path specified by the <code>CY_GETLIBS_SHARED_PATH</code> variable. The default location is one directory level above the current application directory (Default: <code>..</code>). This is used with <code>CY_GETLIBS_SHARED_NAME</code> variable, which specifies the directory name.</p>

~~3 PSoC™ 6 software tools~~

Variable	Description
CY_GETLIBS_SHARED_NAME	<p>Directory name of the shared repo location.</p> <p>All .mtb files have the format, <URI><COMMIT><LOCATION>. If the <LOCATION> field begins with \$\$ASSET_REPO\$\$, then the repo is deposited in the directory specified by the CY_GETLIBS_SHARED_NAME variable. The default directory name is "mtb_shared". This is used with CY_GETLIBS_SHARED_PATH variable, which specifies the directory path.</p>

3.3.3.9.5 Path make variables

These variables are used to specify various paths.

Variable	Description
CY_APP_PATH	<p>Relative path to the top-level of application. For example, ./</p> <p>Setting this path to other than ./ allows the auto-discovery mechanism to search from a root directory location that is higher than the application directory. For example, CY_APP_PATH=.../.../ allows auto-discovery of files from a location that is two directories above the location of ./Makefile.</p>
CY_BASELIB_PATH	<p>Relative path to the base library. For example, ./libs/recipe-make-cat1a</p> <p>This directory must be relative to CY_APP_PATH. It defines the location of the library containing the recipe Makefiles, where the expected directory structure is <CY_BASELIB_PATH>/make. All applications must set the location of the recipe base library. For applications using .mtb files, the BSP's TARGET.mk file sets this variable and therefore the application does not need to.</p>
CY_BASELIB_CORE_PATH	<p>Relative path to the core base library. For example, ./libs/core-make</p> <p>This directory must be relative to CY_APP_PATH. It defines the location of the library containing the core make files, where the expected directory structure is <CY_BASELIB_CORE_PATH>/make. All applications must set the location of the core base library.</p> <p>For applications using .mtb files, the BSP's TARGET.mk file sets this variable and therefore the application does not need to. This variable is not applicable for applications using the combined base library (such as recipe-make-cat1a).</p>
CY_EXTAPP_PATH	<p>Relative path to an external application directory. For example, ../external</p> <p>This directory must be relative to CY_APP_PATH. Setting this path allows incorporating files external to CY_APP_PATH.</p> <p>For example, CY_EXTAPP_PATH=.../external lets auto-discovery pull in the contents of ../external directory into the build.</p>
CY_COMPILER_PATH	<p>Absolute path to the compiler (default: GCC_ARM in cy_TOOLS_DIR).</p> <p>Setting this path allows custom toolchains to be used instead of the defaults. This should be the location of the /bin directory containing the compiler, assembler, and linker. For example:</p> <p>CY_COMPILER_PATH="C:/Program Files (x86)/IAR Systems/Embedded Workbench 8.4/arm/"</p>

~~3 PSoC™ 6 software tools~~

Variable	Description
CY_TOOLS_DIR	<p>Absolute path to the tools root directory.</p> <p>Example Usage: <code>make build CY_TOOLS_DIR="path/to/ModusToolbox/tools_x.y"</code></p> <p>Applications must specify the tools_<version> directory location, which contains the root Makefile and the necessary tools and scripts to build an application.</p> <p>Application Makefiles are configured to automatically search in the standard locations for various platforms. If the tools are not located in the standard location, you may explicitly set this.</p>
CY_BUILD_LOCATION	<p>Absolute path to the build output directory (default: <code>pwd/build</code>).</p> <p>The build output directory is structured as <code>/TARGET/CONFIG/</code>. Setting this variable allows the build artifacts to be located in the directory pointed to by this variable.</p>
CY PYTHON PATH	<p>Specifies the path to a specific Python executable.</p> <p>Example Usage: <code>CY PYTHON PATH="path/to/python/executable/python.exe"</code></p> <p>For make targets that depend on Python, the build system looks for Python 3 in the user's PATH and uses that to invoke python. If you have a version of Python installed in a non-default location and do not have a path set for it, you can set <code>CY PYTHON PATH</code> as a System Variable. In Windows, you must use forward slashes in the path to the Python executable.</p>
CY_DEVICESUPPORT_PATH	<p>Relative path to the devicesupport.xml file.</p> <p>This path specifies the location of the devicesupport.xml file for the Device Configurator. It is used when the configurator needs to be run in a multi-application scenario.</p>
TOOLCHAIN_MK_PATH	<p>Specifies the location of a custom TOOLCHAIN.mk file.</p> <p>Defining this path allows the build system to use a custom TOOLCHAIN.mk file pointed to by this variable.</p>

3.3.3.9.6 Miscellaneous make variables

These are miscellaneous variables used for various make targets.

Variable	Description
CY_IGNORE	<p>Adds to the directory and file ignore list. For example, <code>./file1.c ./inc1</code></p> <p>Example Usage: <code>make build CY_IGNORE="path/to/file/ignore_file"</code></p> <p>Directories and files listed in this variable are ignored in auto-discovery.</p> <p>This mechanism works in combination with any existing .cyignore files in the application.</p>
CY_SKIP_RECIPE	<p>Skip including the recipe Makefiles.</p> <p>Setting this to [true/1] allows the application to not include any recipe Makefiles and only include the start.mk file from the tools install.</p>
CY_SKIP_CDB	<p>Skip creating .cdb files.</p> <p>Constant Database (CDB) files are generated during the build process. Setting this to [true] allows the build process to skip the .cdb files creation.</p>

~~3 PSoC™ 6 software tools~~

Variable	Description
CY_EXTRA_INCLUDES	<p>Specifies additional Makefiles to add to the build.</p> <p>Example Usage: <code>make build CY_EXTRA_INCLUDES=".custom1.mk"</code></p> <p>This variable provides a way of injecting additional make files into the core make files. It can be used when including the make file before or after start.mk in the application Makefile is not possible.</p>
CY_LIBS_SEARCH_DEPTH	<p>Directory search depth for .mtb files (default: 5).</p> <p>Example Usage: <code>make getlibs CY_LIBS_SEARCH_DEPTH=7</code></p> <p>This variable controls how deep the search mechanism in getlibs looks for .mtb files.</p>
CY_UTILS_SEARCH_DEPTH	<p>Directory search depth for .cyignore and TARGET.mk files (default: 5).</p> <p>Example Usage: <code>make getlibs CY_UTILS_SEARCH_DEPTH=7</code></p> <p>This variable controls how deep the search mechanism looks for .cyignore and TARGET.mk files. Min=1, Max=9.</p>
CY_IDE_PRJNAME	<p>Name of the Eclipse IDE application.</p> <p>Example Usage: <code>make eclipse CY_IDE_PRJNAME="AppV1"</code></p> <p>This variable can be used to define the file and target application name when generating Eclipse launch configurations in the eclipse target.</p>
CY_CONFIG_FILE_EXT	<p>Specifies the configurator file extension. For example, *.modus.</p> <p>Example Usage: <code>make get_app_info CY_CONFIG_FILE_EXT="modus cybt cyusbdev"</code></p> <p>This variable accepts a space-separated list of configurator file extensions to search when running the get_app_info target.</p>
CY_SUPPORTED_TOOL_TYPES	<p>Defines the supported tools for a BSP.</p> <p>Example Usage (bsp.mk): <code>CY_SUPPORTED_TOOL_TYPES+=seglcd-configurator</code></p> <p>BSPs can define the supported tools that can be launched using the open target. The supported tool types are bt-configurator, capsense-configurator, capsense-tuner, device-configurator, dfuh-tool, library-manager, project-creator, qspi-configurator, seglcd-configurator, smartio-configurator, and usbdev-configurator. The BSP can make adjustments to the default recipe if needed.</p>

3 PSoC™ 6 software tools**3.3.4 Board support packages****3.3.4.1 Overview**

A BSP provides a standard interface to a board's features and capabilities. The API is consistent across our kits. Other software (such as middleware or an application) can use the BSP to configure and control the hardware. BSPs do the following:

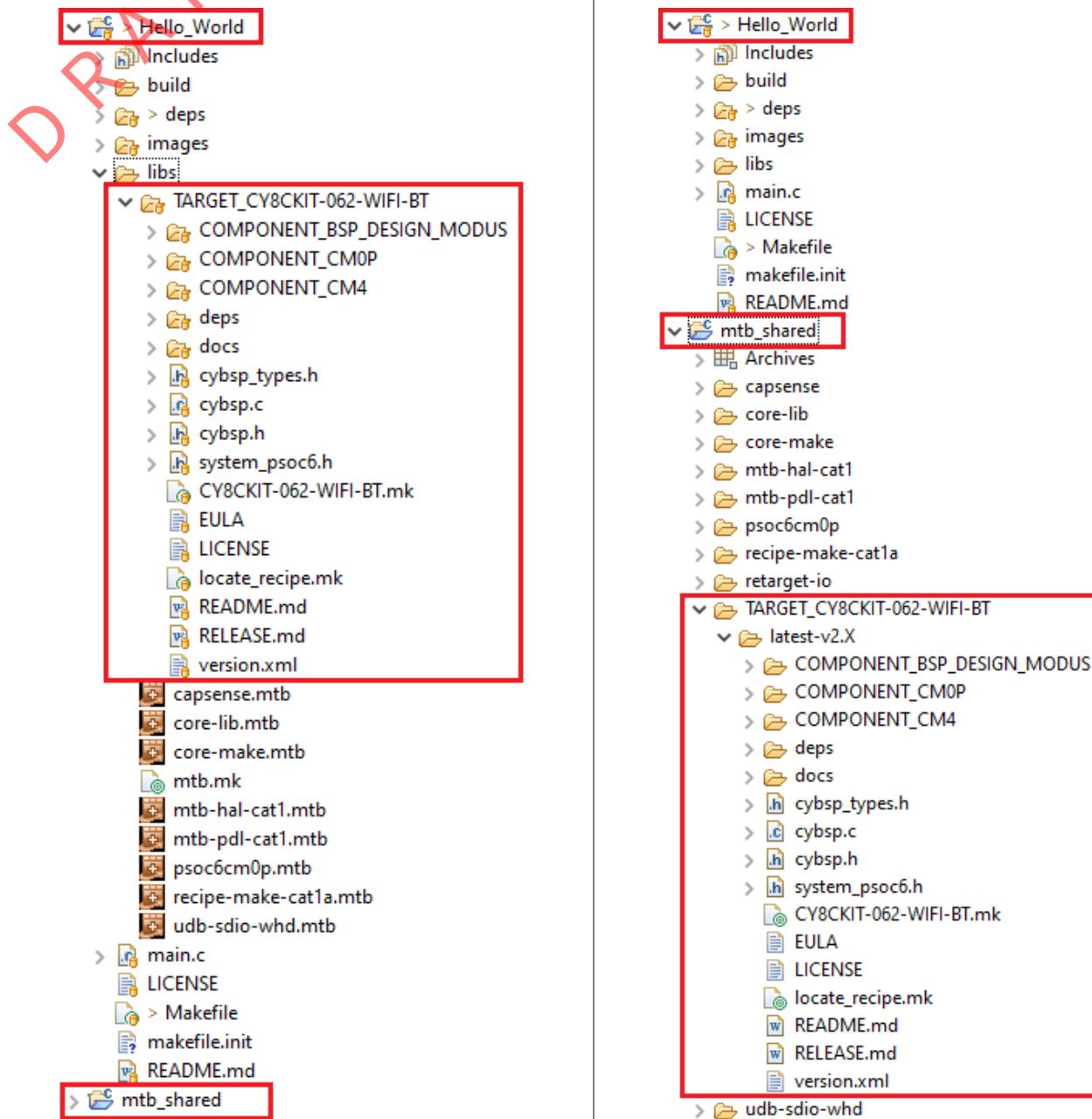
- initialize device resources, such as clocks and power supplies to set up the device to run firmware.
- contain default linker scripts and startup code that you can customize for your board.
- contain the hardware configuration (structures and macros) for both device peripherals and board peripherals.
- provide abstraction to the board by providing common aliases or names to refer to the board peripherals, such as buttons and LEDs.
- include the libraries for the default capabilities on the board. For example, the BSP for a kit with CAPSENSE™ capabilities includes the CAPSENSE™ library.

3.3.4.2 What's in a BSP

This section presents an overview of the key resources that are part of a BSP. Using the MTB flow, applications can share BSPs and libraries. BSPs that are local to the application are located in the libs subdirectory, while shared BSPs are located in the mtb_shared directory adjacent to the application directory. For more details about library management, refer to the [Library Manager user guide](#).

The following shows a typical PSoC™ 6 BSP located in the application's libs subdirectory on the left. It also shows a shared BSP located in the mtb_shared directory on the right.

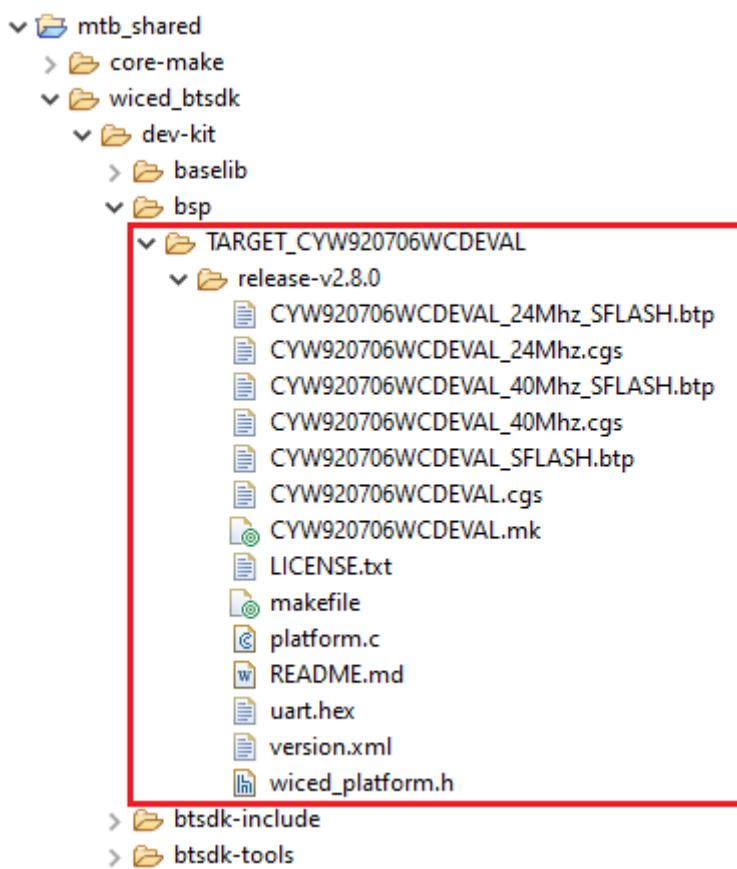
3 PSoC™ 6 software tools



Note: For BTSDK v2.8 and later, shared BSPs and some shared libraries are located in subdirectories in the mtb_shared directory. For example:

3 PSoC™ 6 software tools

DRAFT



For BTSDK v2.7 and earlier, shared BSPs and libraries can be found in the same structure, but without the leading mtb_shared directory as shown in the previous image.

The following sections describe the various files and directories in a typical BSP:

3.3.4.2.1 COMPONENT_BSP_DESIGN_MODUS

This directory contains the configuration files (such as design.modus) for use with various BSP configurator tools, including 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. See [Modifying the BSP configuration for a single application](#) to learn how the application can override this component.

3.3.4.2.2 COMPONENT

Some applications may have additional "COMPONENT" subdirectories. These directories are conditional, based on what the BSP is being built for. For example, the PSoC™ 6 BSPs include COMPONENT directories to restrict which files are used when building for the Arm Cortex M4 or M0+ core.

3.3.4.2.3 deps subdirectory

The deps subdirectory inside the BSP contains .lib files from earlier versions of ModusToolbox™. This is not the same as the deps subdirectory inside the application that contains the .mtb files. See [Typical application contents](#) for more details.

3.3.4.2.4 docs subdirectory

The docs subdirectory contains the documentation in HTML format for the selected BSP.

~~3 PSoC™ 6 software tools~~

~~3.3.4.2.5~~ Support files

DRAFT
Different BSPs will contain various files, such as the API interface to the board's resources. For example, a typical PSoC™ 6 BSP contains the following:

- cybsp.c / .h – You need to include only cybsp.h in your application to use all the features of a BSP. Call cybsp_init() from cybsp.c to initialize the board.
- cybsp_types.h – This currently contains Doxygen comments. It is intended to contain the aliases (macro definitions) for all the board resources, as needed.
- system_psoc6.h – This file provides information about the chip initialization that is done pre-main().

3.3.4.2.6 <BSP_NAME>.mk

This file defines the DEVICE and other BSP-specific make variables such as COMPONENTS. These are described in the [ModusToolbox™ build system](#) chapter. It also defines board-specific information such as the device ID, compiler and linker flags, pre-builds/post-builds, and components used with this board implementation.

3.3.4.2.7 locate_recipe.mk

This is a helper file for the BSP to specify the path to the core and recipe Makefiles that are included as dependent libraries.

3.3.4.2.8 README/RELEASE.md

These are documentation files. The README.md file describes the BSP overall, while the RELEASE.md file covers changes made to version of the BSP.

3.3.4.2.9 BTSDK-specific BSP files

BTSDK BSPs may optionally provide the following types of files:

- wiced_platform.h – Platform specific structures to define hardware information such as max number of GPIOs, LEDs or user buttons available
- Makefile – Provided to support LIB flow applications (BTSDK 2.7 and earlier). Not used in MTB flow BTSDK 2.8 or later applications.
- *.hex – binary application image files that are used as part of the embedded application creation, program, and/or OTA (Over-The-Air) upgrade processes.
- platform*.c/h – Platform specific source and header files used by platform and application initialization functions.
- <BSP_NAME>*.cgs – Patch configuration records in text format, can be multiple copies supporting various board configurations.
- <BSP_NAME>*.btp – Configuration options related to building and programming the application image, can be multiple copies supporting various board configurations.

3.3.4.3 Creating your own BSP

This section contains a condensed version of these instructions. For a better understanding of the contents and structure of a BSP and more detailed information about how to update the Wi-Fi and Bluetooth® connectivity device and firmware in a BSP, refer to <https://www.cypress.com/ModusToolboxCreateCustomBSP>.

In most cases before you do any real design work on your application, you should create a BSP for your device and/or board. This allows you to configure the settings for your own custom hardware or for different linker options. Plus, you can save the BSP for use in future applications.

~~3 PSoC™ 6 software tools~~

There are two basic methods to create a BSP; each involves creating an application. Using the first method, specify the closest-matching BSP to your intended BSP. In this case, you usually have to remove various settings and options that you don't need. For the second method, specify a "generic" BSP template when creating your application. In this case, your BSP is essentially built from scratch, and you need to add and configure settings and options for your needs.

Regardless of the method you choose, the basic process is the same for both:

1. Create a simple example application. Use a BSP that is close to your goal or select a "generic" BSP.
2. Navigate to the application directory, and run the make bsp target. Specify the new board name by passing the value to the TARGET_GEN variable. This is the minimum required. For example, to create a BSP called MyBSP:

```
make bsp TARGET_GEN=MyBSP
```

Optionally, you may use DEVICE_GEN to specify a new device if it is different than the one included with the original BSP. For example:

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

The make bsp 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 make getlib process automatically creates indirect dependencies for .mtbx files in custom BSPs.

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's deps subdirectory. Be sure to remove the corresponding .mtbx files as well.

3. Update the application's Makefile TARGET variable to point to your new BSP. For example:

```
TARGET=MyBSP
```

4. Open the Device Configurator to customize settings in the new BSP's design.modus file for pin names, clocks, power supplies, and peripherals as required. Also, address any issues that arise.
5. Start writing code for your application.

If using an IDE, you need to generate/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
```

Note: Use make help to see all supported IDE make targets. See also the [Exporting to supported IDEs](#) chapter 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 the [Manifest files](#) chapter 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.

~~3 PSoC™ 6 software tools~~

~~3.3.4.4 Modifying the BSP configuration for a single application~~

In cases where you don't want to create a BSP, you can modify the BSP configuration for a single application (such as different pin or peripheral settings). However, you should not typically modify the BSP directly since that results in changes to the BSP library. This would prevent you from updating the repository in the future, and it may affect other applications in the same workspace. Instead, use the following process to create a custom set of configuration files for a specific application:

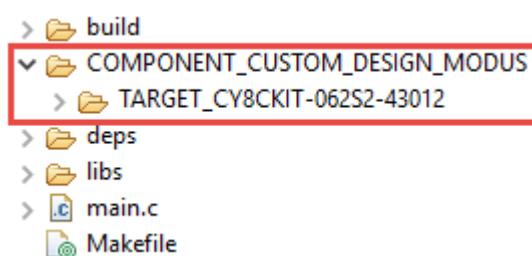
1. Create a directory at the root of the application to hold any custom BSP configuration files. For example:
Hello_World/COMPONENT_CUSTOM_DESIGN_MODUS

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.



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.

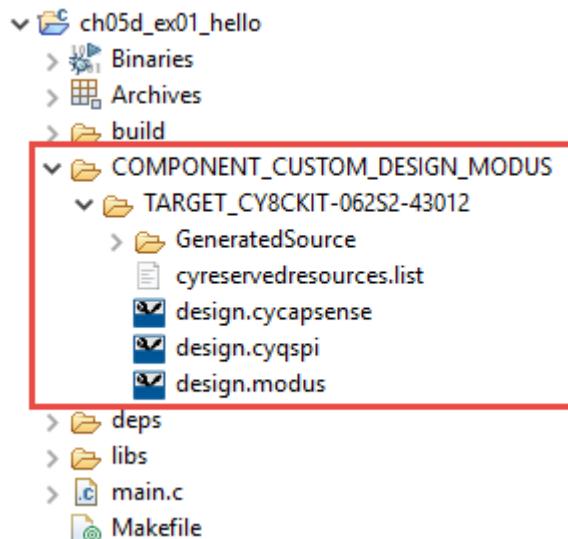
The second line is required to specify the new directory to include your custom configuration files, and to ensure that the init_cycfg_all function is still called from the cybsp_init function. The init_cycfg_all function is used to initialize the hardware that was set up in the configuration files.

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.*

3 PSoC™ 6 software tools

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:



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

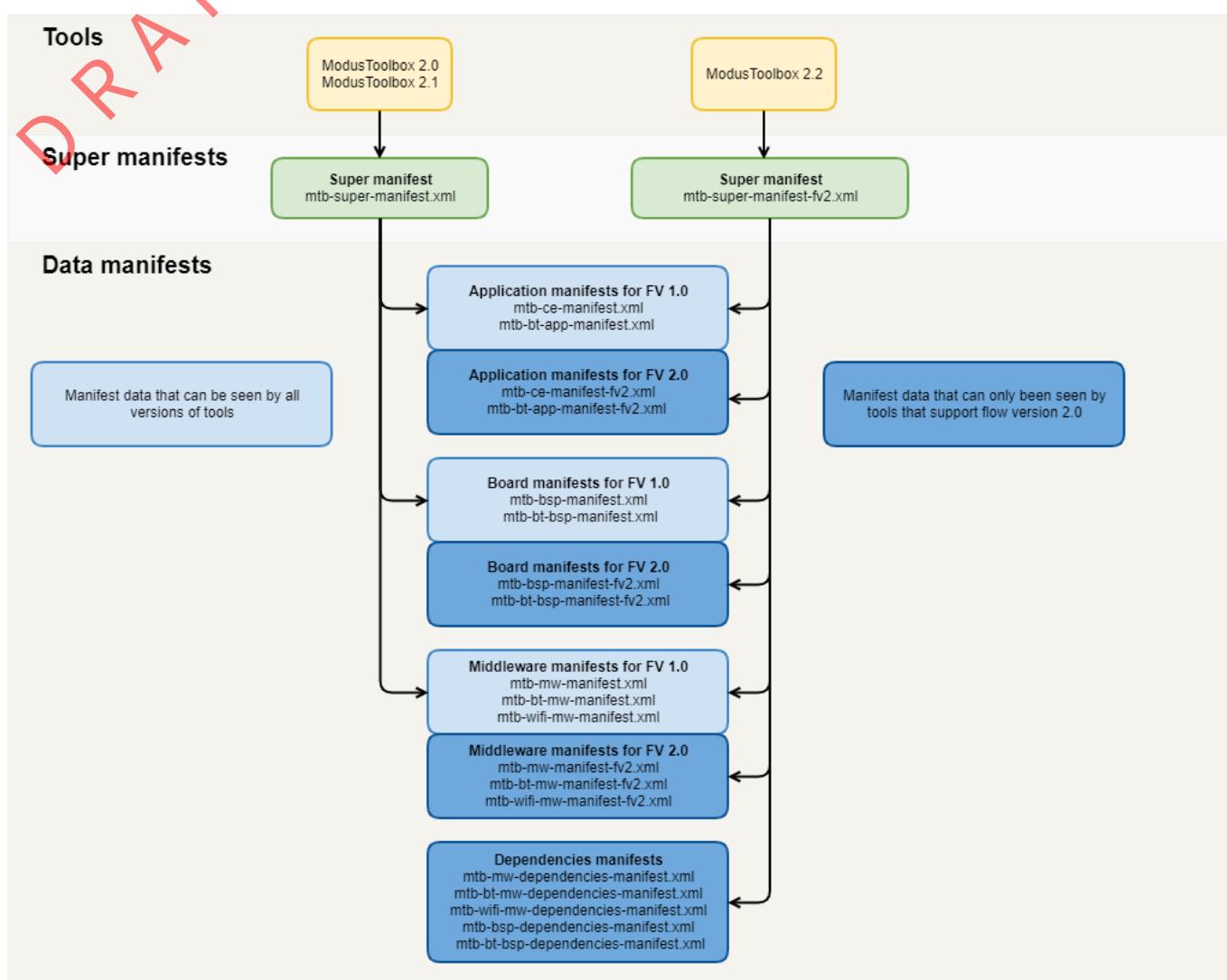
3 PSoC™ 6 software tools**3.3.5 Manifest files****3.3.5.1 Overview**

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.

Beginning with the ModusToolbox™ 2.2 release, there are two versions of manifest files: the existing ones for the LIB flow and earlier versions of ModusToolbox™ software, and new ones for the MTB flow (aka "fv2"). The existing super-manifest file for use with the ModusToolbox™ 2.1 release and earlier contains only references to manifests that contain items that support the LIB flow. The new super-manifest file for use with the ModusToolbox™ 2.2 release and later contains references to all the manifest files.

3 PSoC™ 6 software tools



3.3.5.2 Create your own manifest

By default, the ModusToolbox™ tools look for our manifest files maintained on our server. So, the initial lists of BSPs, code examples, and middleware available to use are limited to our manifest files. You can create your own manifest files on your servers or locally on your machine, and you can override where ModusToolbox™ tools look for manifest files.

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

- Super-manifest: <https://github.com/Infineon/mtb-super-manifest>
- Code example manifest: <https://github.com/Infineon/mtb-ce-manifest>
- BSP manifest (including dependencies): <https://github.com/Infineon/mtb-bsp-manifest>
- Middleware manifest (including dependencies): <https://github.com/Infineon/mtb-mw-manifest>

Make sure you look at the "fv2" manifest files if you are using the MTB flow.

The manifest system is flexible, and there are multiple paths you can follow to customize the manifests.

- You can customize a super-manifest file and override the default file used by the tools.

~~3 PSoC™ 6 software tools~~

- You can create secondary super-manifest files that identify additional content. The tools will merge your additional content with the default super-manifest.
- You can modify or replace any of the default manifest files (code example, BSP, etc.) with custom files, so long as your custom super-manifest file points to those rather than the default files.

3.3.5.2.1 Overriding the standard super-manifest

The location of the standard super-manifest file is hard coded into all of the tools. However, you may override this location by setting the CyRemoteManifestOverride environment variable. When this variable is set, the tools use the value of this variable as the location of the super-manifest file and the hard-coded location is ignored. For example:

```
CyRemoteManifestOverride=https://myURL.com/mylocation/super-manifest.xml
```

3.3.5.2.2 Secondary super-manifest

In addition to the standard super-manifest file, you can specify additional super-manifest files. This allows you to add additional items (BSPs, code examples, libraries) along with the standard items. Do this by creating a file called manifest.loc in a hidden directory in your home directory named ".modustoolbox."

<user_home>/modusToolbox/manifest.loc

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: *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*

If the manifest.loc file exists, then each line in this file is treated as the URL to a super-manifest file. These are called the secondary or custom super-manifest files. The format of these files is exactly like the standard super-manifest file. Each of the custom super-manifest files are treated as separate super-manifest files. See the [Conflicting data](#) section for dealing with conflicts.

3.3.5.2.3 Processing

The process for using the manifest files is the same for all tools that use the data. The first step is to access the super-manifest file(s) to obtain a list of manifest files for each of the categories that the tool cares about. For example, the Library Manager tool cares about the board and middleware manifests.

The second step is to retrieve the manifest data from each manifest file and merge the data into a single global data model in the tool. See the [Conflicting data](#) section for dealing with conflicts.

There is no per-file scoping. All data is merged before it is presented. The combination of a super manifest file and the merging of all of the data allows various contributors, including third party contributors, to make new data available without requiring coordinated releases between the various contributors.

The following table shows how manifests are processed:

Source	Syntax	Effect
CyRemoteManifestOverride	valid URL (e.g., file:/// ... or http:// ...)	Use that URL to fetch the super-manifest.

3 PSoC™ 6 software tools

DRAFT

Source	Syntax	Effect
manifest.loc	Fragment (e.g., my/manifests/super-manifest.xml)	Append the home directory to the front (e.g., file:///c:/Users/benh/my/manifests/super-manifest.xml)
	valid URL (e.g., file:/// ... or http:// ...)	Use that URL to fetch the super-manifest.
Manifest URLs	valid URI (e.g., file:/// ... or http:// ...)	Use that URI to fetch the manifest.
Manifest URLs from a local super-manifest file	fragment (e.g., my/manifests/manifest.xml)	Append the directory in which source super-manifest resides (e.g., file:///c:/Users/benh/.modustoolbox/my/manifests/manifest.xml)
Manifest URLs from a remote super-manifest file	fragment (e.g., my/manifests/manifest.xml)	Append the home directory to the front (e.g., file:///c:/Users/benh/my/manifests/manifest.xml)

3.3.5.2.4 Conflicting data

Ultimately, data from all of the super-manifest and manifest files are combined into a single data collection of BSPs, code examples, and middleware. During the collation of this data, there may be conflicting data entries. There are two types of conflicts.

The first kind is a conflict between data that comes from the primary super-manifest (and linked manifests) and data that comes from the custom super-manifest (and linked manifests). In this case, the data in the custom location overwrites the data from the standard location. This mechanism allows you to intentionally override data that is in the standard location. In this case, no error or warning is issued. It is a valid use case.

The second kind of conflict is between data coming from the same source (that is, both from primary or both from secondary). In this case, an error message is printed and all pieces of conflicting data are removed from the data model. This is done because in this case, it is not clear which data item is the correct one.

3.3.5.3 Using offline content

In normal mode, ModusToolbox™ tools look for manifest files maintained on GitHub and downloads the firmware libraries from git repositories referenced by the manifests. If a network connection to online resources is not available, you can download a copy of all manifests and content, and then point the tools to use this copy in offline mode. This section describes how to download, install, and use the offline content.

Note: *ModusToolbox™ libraries are updated frequently, and the offline content does not always have the latest versions available. We strongly recommend using the online content whenever possible. See <https://community.cypress.com/docs/DOC-19903> for more details.*

1. Download the modustoolbox-offline-content.zip file from our website:
<https://www.cypress.com/products/modustoolbox-software-environment>

~~3 PSoC™ 6 software tools~~

- ~~DRAFT~~
2. If you do not already have a hidden directory named .modustoolbox in your home directory, create one. Using Cygwin on Windows for example:

```
mkdir -p "$USERPROFILE/.modustoolbox"
```

3. Extract the ZIP archive to the ./modustoolbox sub-directory in your home directory. The resulting path should be:

~/modustoolbox/offline

The following is a Cygwin on Windows command-line example to use for extracting the content:

```
unzip -qbd "$USERPROFILE/.modustoolbox" modustoolbox-offline-content.zip
```

Note: If you previously installed a copy of the offline content, you should delete the existing ~/modustoolbox/offline directory before extracting the archive. Using Cygwin on Windows for example:

```
rm -rf "$USERPROFILE/.modustoolbox/offline"
```

4. To use the Project Creator GUI or Library Manager GUI in offline mode, select Offline from the Settings menu (refer to the appropriate user guide for details).

Note: Make sure CyRemoteManifestOverride variable is not set when you use offline mode.

5. To use the Project Creator CLI in offline mode, execute the tool with the --offline argument. For example:

```
project-creator-cli --board-id CY8CPROTO-062-4343W --app-id mtb-example-psoc6-hello-world  
--offline
```

6. The Project Creator and Library Manager tools execute the make getlibs command under the hood to download/update the firmware libraries. To execute the make getlibs target in offline mode, pass the CY_GETLIBS_OFFLINE=true argument:

```
make getlibs CY_GETLIBS_OFFLINE=true
```

To override the location of the offline content, set the CY_GETLIBS_OFFLINE_PATH variable:

```
make getlibs CY_GETLIBS_OFFLINE=true CY_GETLIBS_OFFLINE_PATH="~/custom/offline/content"
```

Refer to the [ModusToolbox™ build system](#) chapter for more details about make targets and variables.

7. Once network connectivity is available, you can use the Library Manager tool to update the ModusToolbox™ project previously created offline to use the latest available content. Or you can execute the make getlibs command **without** the CY_GETLIBS_OFFLINE argument.

3.3.5.4 Access private repositories

You can extend the custom manifest with additional content from git repositories (repos) hosted on GitHub or any other online git server. To access private git repos, you must configure the git client so that the

3 PSoC™ 6 software tools

Project Creator and Library Manager tools can authenticate over HTTP/HTTPS protocols without an interactive password prompt.

Note: While you can host libraries on private repos, the custom content manifest must be accessible without authentication (that is, it cannot be hosted on a private git repo).

To configure git credentials for non-interactive remote operations over HTTP protocols, refer to the git documentation:

- <https://git-scm.com/book/en/v2/Git-Tools-Credential-Storage>
- <https://git-scm.com/docs/git-credential-store>

The simplest way is to configure a git-credential-store and save the HTTP credentials in a plain text file. Note that this option is less secure than other git credential helpers that use OS credentials storage.

The following steps show how to configure a git client to access GitHub private repositories without a password prompt:

1. Login to GitHub and go to Personal access tokens: <https://github.com/settings/tokens>
2. Click Generate new token to open the New personal access token screen.
3. On that screen:
4. Type some text in the Note field.
5. Under Select scopes, click on repo.
6. Click Generate token (scroll down to see the button).
7. Copy the generated token.
8. Open an interactive shell (for example, modus-shell\Cygwin.bat on Windows), and type the following commands (replace the user name and token with your information):

```
git config --global credential."https://github.com".helper store
GITHUB_USER=<your-github-username>
GITHUB_TOKEN=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx # generated at https://github.com/settings/
tokens
echo "https://$GITHUB_USER:$GITHUB_TOKEN@github.com" >> ~/.git-credentials
```

After entering the commands, you can clone private GitHub repositories without an interactive user/password prompt.

Note: A GitHub account password can be used instead of GITHUB_TOKEN, in case the 2FA (two-factor authentication) is not enabled for the GitHub account. However, this option is not recommended.

~~3 PSoC™ 6 software tools~~

~~3.3.6 Using applications with third-party tools~~

ModusToolbox™ software includes a variety of ways to use applications with third-party tools. This chapter covers the following:

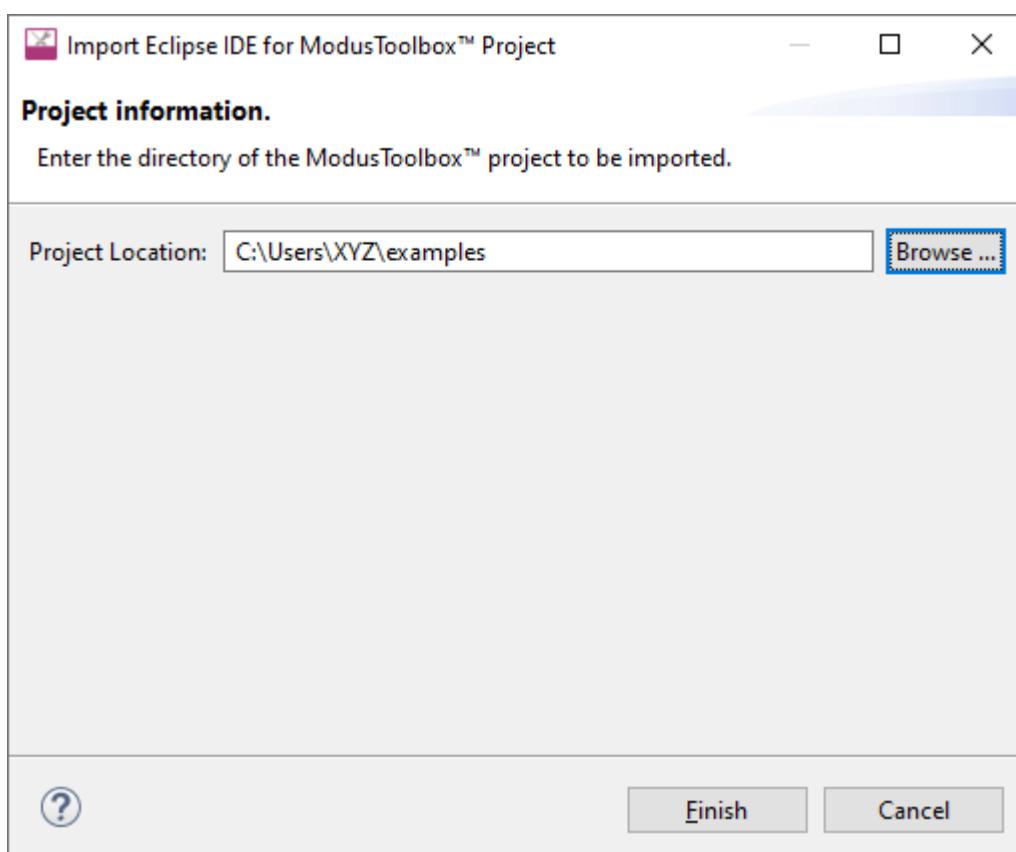
- Import to Eclipse
- Exporting to supported IDEs
- Patched flashloaders for AIROC™ CYW208xx devices
- Generating files for XMC™ Simulator tool

3.3.6.1 Import to Eclipse

The easiest way to create a ModusToolbox™ application for Eclipse is to use the Eclipse IDE included with the ModusToolbox™ software. ModusToolbox™ includes an Eclipse plugin that provides links to launch the Project Creator tool and then import the application into Eclipse. For details, refer to the [Eclipse IDE for ModusToolbox™ quick start guide](#) or [user guide](#).

If you already have a ModusToolbox™ application created some other way than through the included Eclipse IDE, you can import that application for use in Eclipse as follows:

1. Open the Eclipse IDE included with ModusToolbox™, and select Import Application on the Quick Panel
2. In the Project Location field, click the Browse... button; navigate to and select the application's directory.



3. Click Finish.

The application displays in the Eclipse IDE Project Explorer.

~~3 PSoC™ 6 software tools~~

~~3.3.6.2 Exporting to supported IDEs~~

~~3.3.6.2.1 Overview~~

This chapter describes how to export a ModusToolbox™ application to various supported IDEs in addition to the provided Eclipse IDE. As described [Getting started](#) chapter, the Project Creator tool includes a Target IDE option that generates files for the selected IDE. Also, as noted in the [ModusToolbox™ build system](#) chapter, the make command includes various targets for the following IDEs:

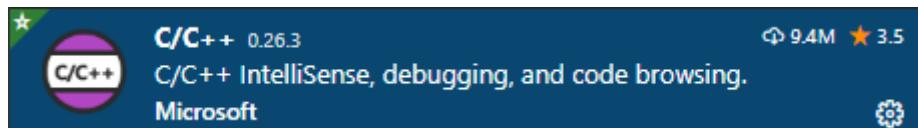
- [Export to VS Code](#): `make vscode`
- [Export IAR EWARM \(Windows only\)](#): `make ewarm8 TOOLCHAIN=IAR`
- [Export to Keil µVision 5 \(Windows only\)](#): `make uvision5 TOOLCHAIN=ARM`

3.3.6.2.2 Export to VS Code

This section describes how to export a ModusToolbox™ application to VS Code.

Prerequisites

- ModusToolbox™ 2.4 software and application
- VS Code version 1.42.x or later
- VS Code extensions. Install the following:
 - c/c++ tools



- Cortex Debug



Note: These versions change often; use the most current.

- For J-Link debugging, download and install J-Link software:
<https://www.segger.com/downloads/jlink>

Process example

1. Create a ModusToolbox™ application.
 - a. If you use the Project Creator tool, choose "VS Code" from the Target IDE pull down menu.
 - b. If you use the command line, open an appropriate shell program (see [CLI set-up instructions](#)), and navigate to the application directory, and run the following command:

```
make vscode
```

Either process generates json files for debug/program launches, IntelliSense, and custom tasks.

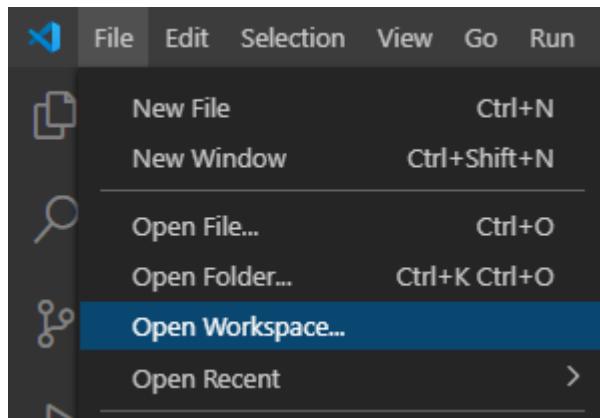
3 PSoC™ 6 software tools

Note:

Any time you update/patch the tools for your application(s), that path information might change. So, you will need to regenerate the needed support files by running the make vscode command or update them manually.

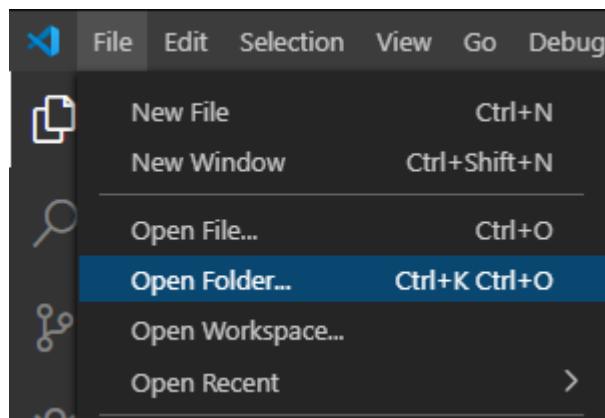
2. Open the VS Code tool.

- To open the application and the mtb_shared directory in the same workspace, select **File > Open Folder...**



- Navigate to the application directory and select the <application_name>.code-workspace file.
- If you have several applications in the workspace, you can add them using **Add workspace folder...**

- To open just the application and select **File > Open Workspace...**



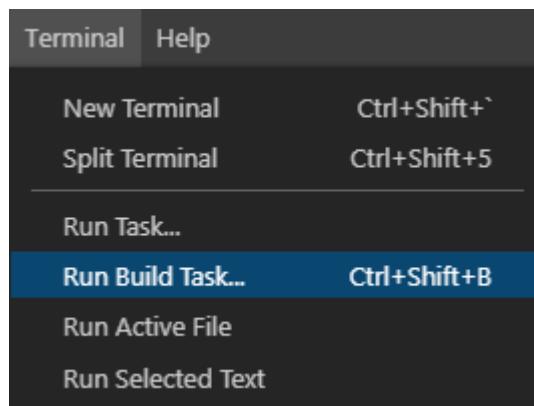
Note: On macOS, this command is File > Open...

Navigate to and select the application directory, and then click **Select Directory**.

- When your application opens in the VS Code IDE, select **Terminal > Run Build Task...**

3 PSoC™ 6 software tools

DRAFT



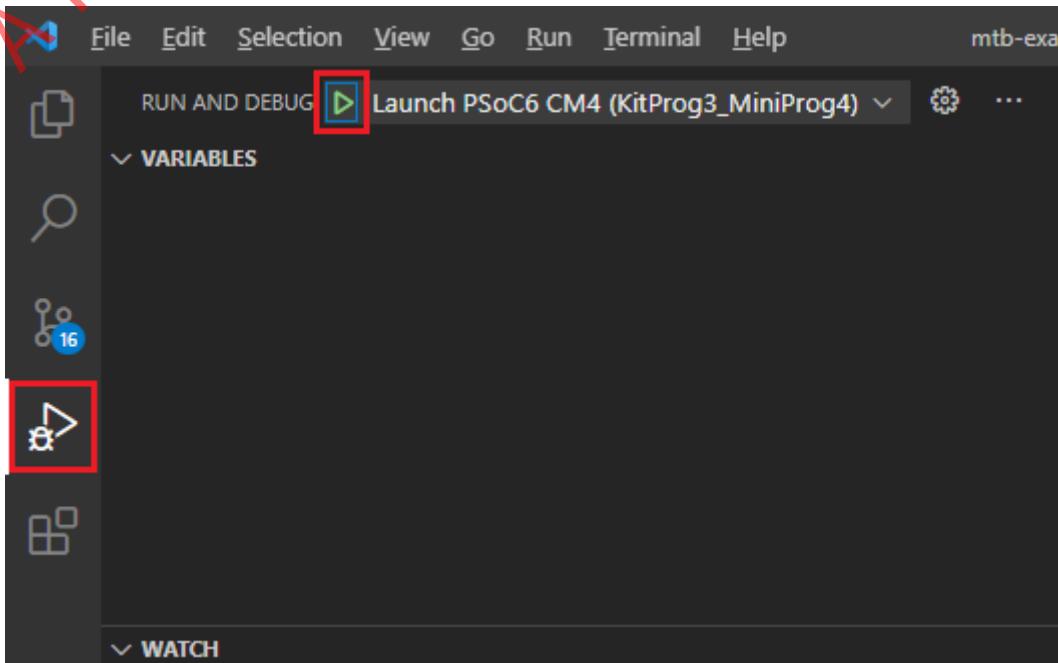
4. Then, select **Build: Build [Debug]**. After building, the VS Code terminal should display messages similar to the following:

To debug using KitProg3/MiniProg4

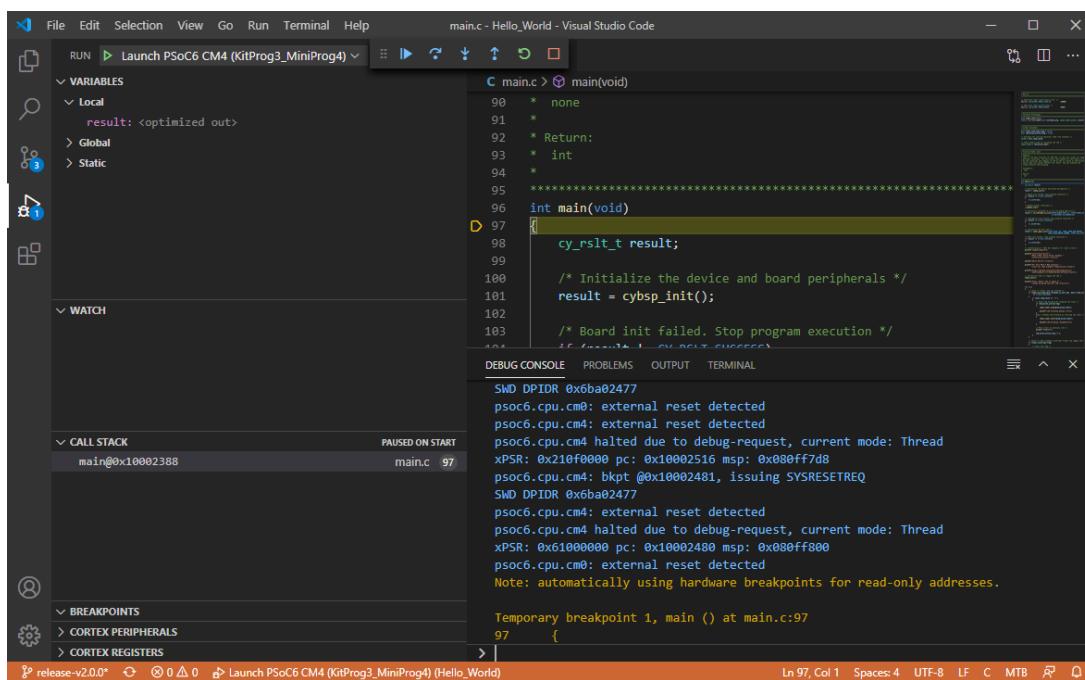
Click the **Run and Debug** icon on the left and then click the **Play** button.

3 PSoC™ 6 software tools

DRAFT



The VS Code tool runs in debug mode.



To debug using J-Link

You can use a J-Link debugger probe to debug the application.

1. Navigate to and open the global settings.json file. If there is no such file, then create one. The file is located here by default:

- Windows: %APPDATA%/Code/User/settings.json
- macOS: \$HOME/Library/Application Support/Code/User/settings.json
- Linux: \$HOME/.config/Code/User/settings.json

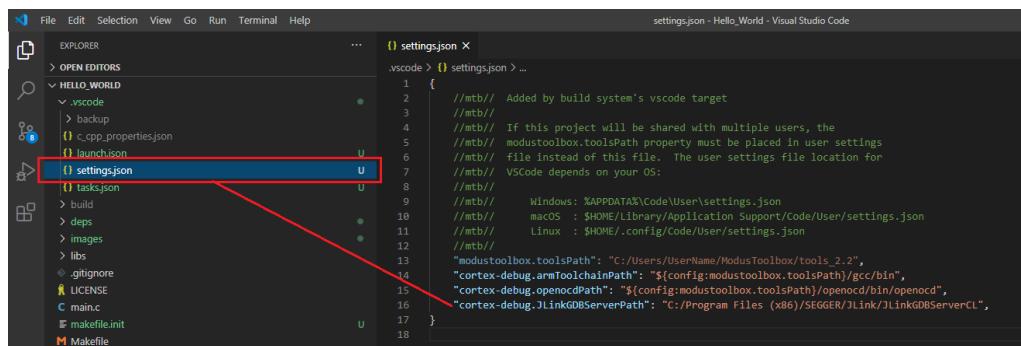
3 PSoC™ 6 software tools

- ~~DRAFT~~
2. Add the path to the J-Link GDB server. For example:

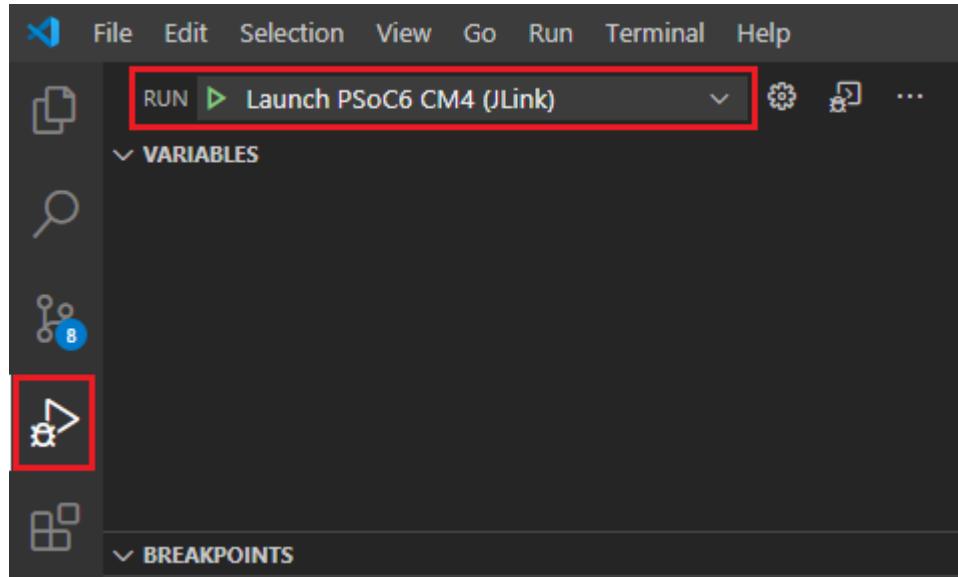
```
{"cortex-debug.JLinkGDBServerPath": "C:/Program Files (x86)/SEGGER/JLink/JLinkGDBServerCL"}
```

- Windows: "cortex-debug.JLinkGDBServerPath": "<JLinkInstallDir>/JLinkGDBServerCL"
- macOS/Linux: "cortex-debug.JLinkGDBServerPath": "<JLinkInstallDir>/JLinkGDBServer"

Note: *The J-Link path can be configured in the local application's settings, if needed.*



3. Click the **Run and Debug** icon, select **Launch PSOC6 CM4 (JLink)** config, and click the **Play** button.



3.3.6.2.3 Export IAR EWARM (Windows only)

This section describes how to export a ModusToolbox™ application to IAR Embedded Workbench and debug it with CMSIS-DAP or J-Link.

Prerequisites

- ModusToolbox™ 2.4 software and application
- Python 3.7 is installed in the tools_2.4 directory, and the make build system has been configured to use it.
You don't need to do anything if you use the modus-shell/Cygwin.bat file to run command line tools.

However, if you plan to use your own version of Cygwin or some other type of bash, you will need to ensure your system is configured correctly to use Python 3.7. Use the CY_PYTHON_PATH as appropriate.

~~3 PSoC™ 6 software tools~~

- IAR Embedded Workbench version 8.42.2 or later
- PSoC™ 6 Kit (for example, CY8CPROTO-062-4343W) with KitProg3 FW
- For J-Link debugging, download and install J-Link software:
https://www.segger.com/downloads/jlink/JLink_Windows.exe

Process example

1. Create a ModusToolbox™ application.
 - a. If you use the Project Creator tool, choose "IAR" from the **Target IDE** pull down menu.
 - b. If you use the command line, open an appropriate shell program (see [CLI set-up instructions](#)), navigate to the application directory, and run the following command:

```
make ewarm8 TOOLCHAIN=IAR
```

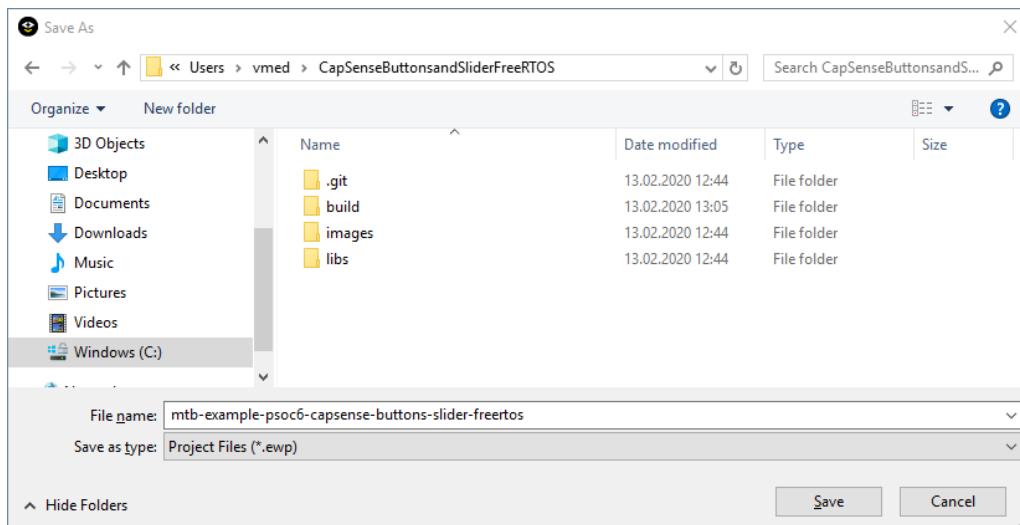
Note: This sets the **TOOLCHAIN** to **IAR** in the **Embedded Workbench configuration files** but **not** in the **ModusToolbox™ application's Makefile**. Therefore, builds inside **IAR Embedded Workbench** will use the **IAR toolchain** while builds from the **ModusToolbox™ environment** will continue to use the **toolchain that was previously specified in the Makefile**. You can edit the Makefile's **TOOLCHAIN** variable if you also want **ModusToolbox™** builds to use the **IAR toolchain**.

Note: Check the output log for instructions and information about various flags.

An IAR connection file appears in the application directory. For example:

mtb-example-psoc6-capsense-buttons-slider-freertos.ipcf

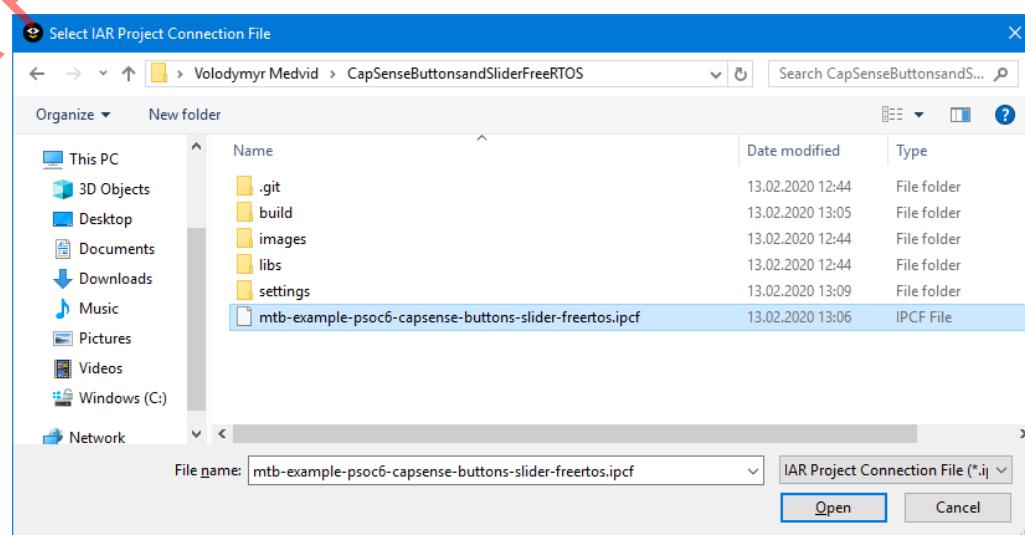
2. Start IAR Embedded Workbench.
3. On the main menu, select **Project > Create New Project > Empty project** and click **OK**.
4. Browse to the **ModusToolbox™** application directory, enter a desired application name, and click **Save**.



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

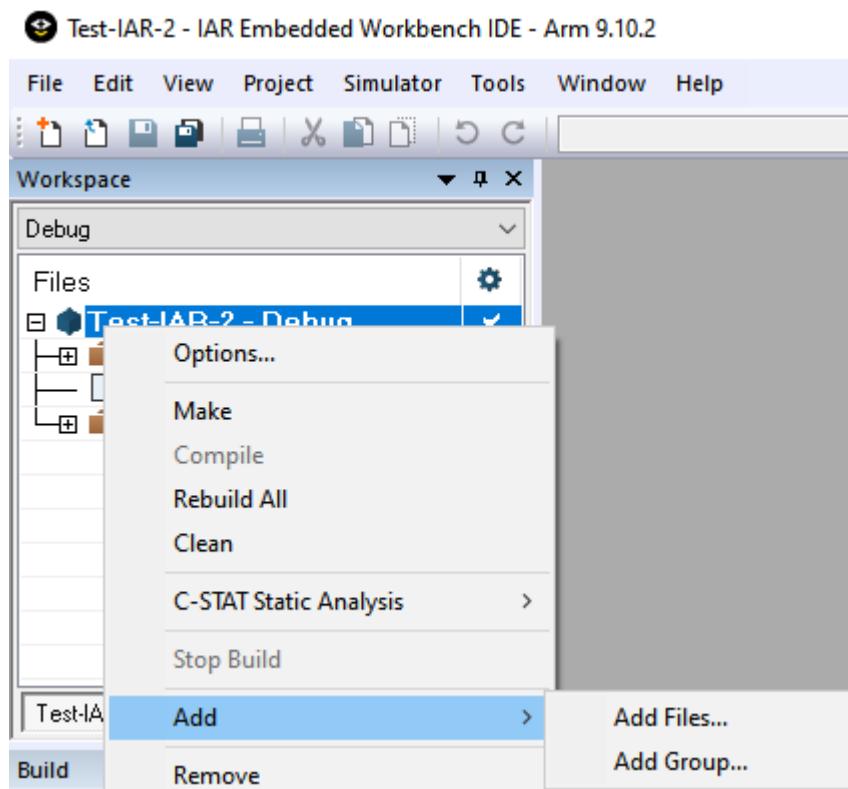
3 PSoC™ 6 software tools

DRAFT



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

Note: If you don't care about staying connected to the ModusToolbox™ tools that generate the project files, you can delete the .ipcf file from the workspace and restart IAR. The official IAR site discusses this option: <https://github.com/IARSystems/project-migration-tools>. If you don't remove the .ipcf file, you need to make all file/group additions at the workspace level.



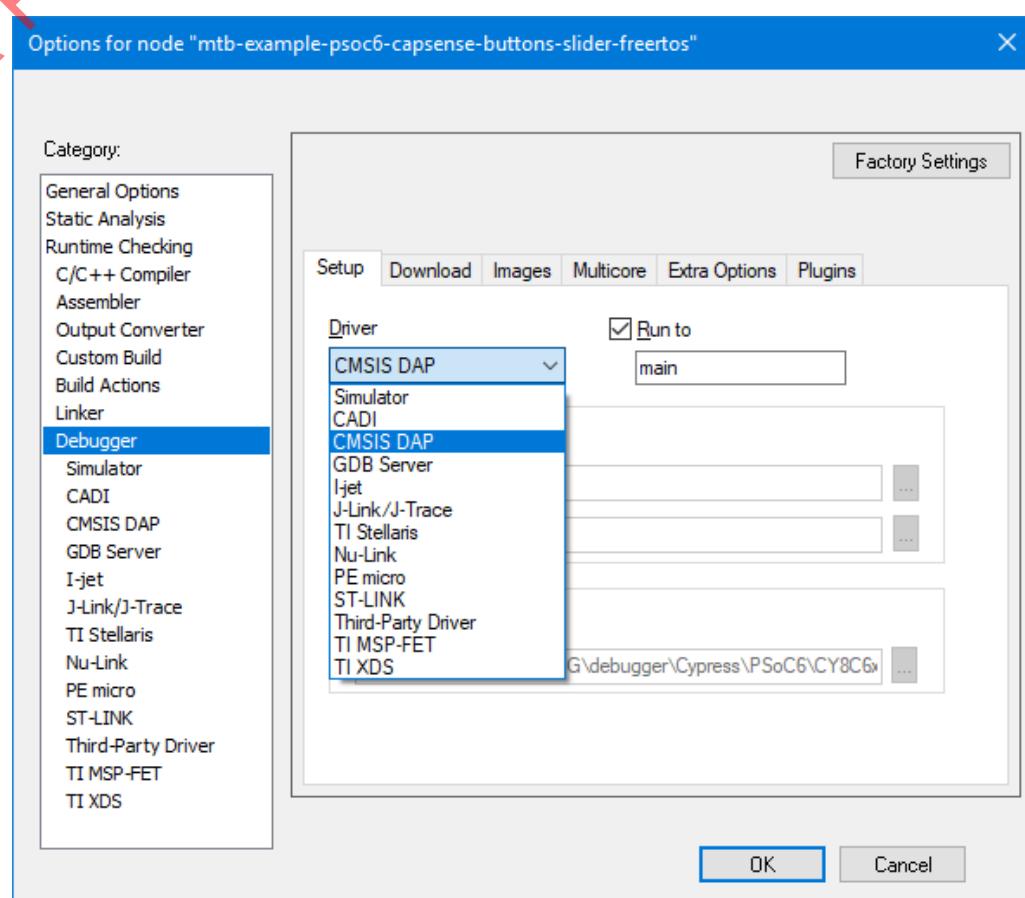
9. Connect the PSoC™ 6 kit to the host PC.

To use KitProg3/MiniProg4

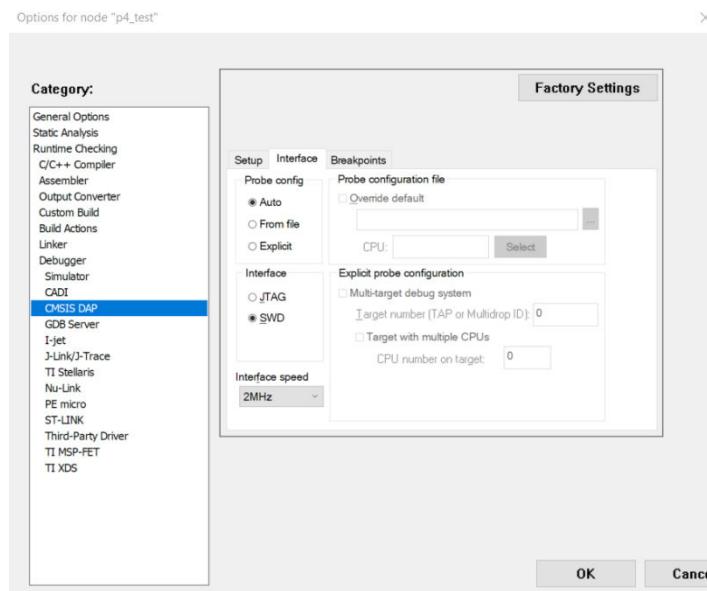
- As needed, run the fw-loader tool to make sure the board firmware is upgraded to KitProg3. See the for details. The tool is in the following directory by default:
`<user_home>/ModusToolbox/tools_2.4/fw-loader/bin/`
- Select **Project > Options > Debugger** and select **CMSIS-DAP** in the Driver list:

3 PSoC™ 6 software tools

DRAFT



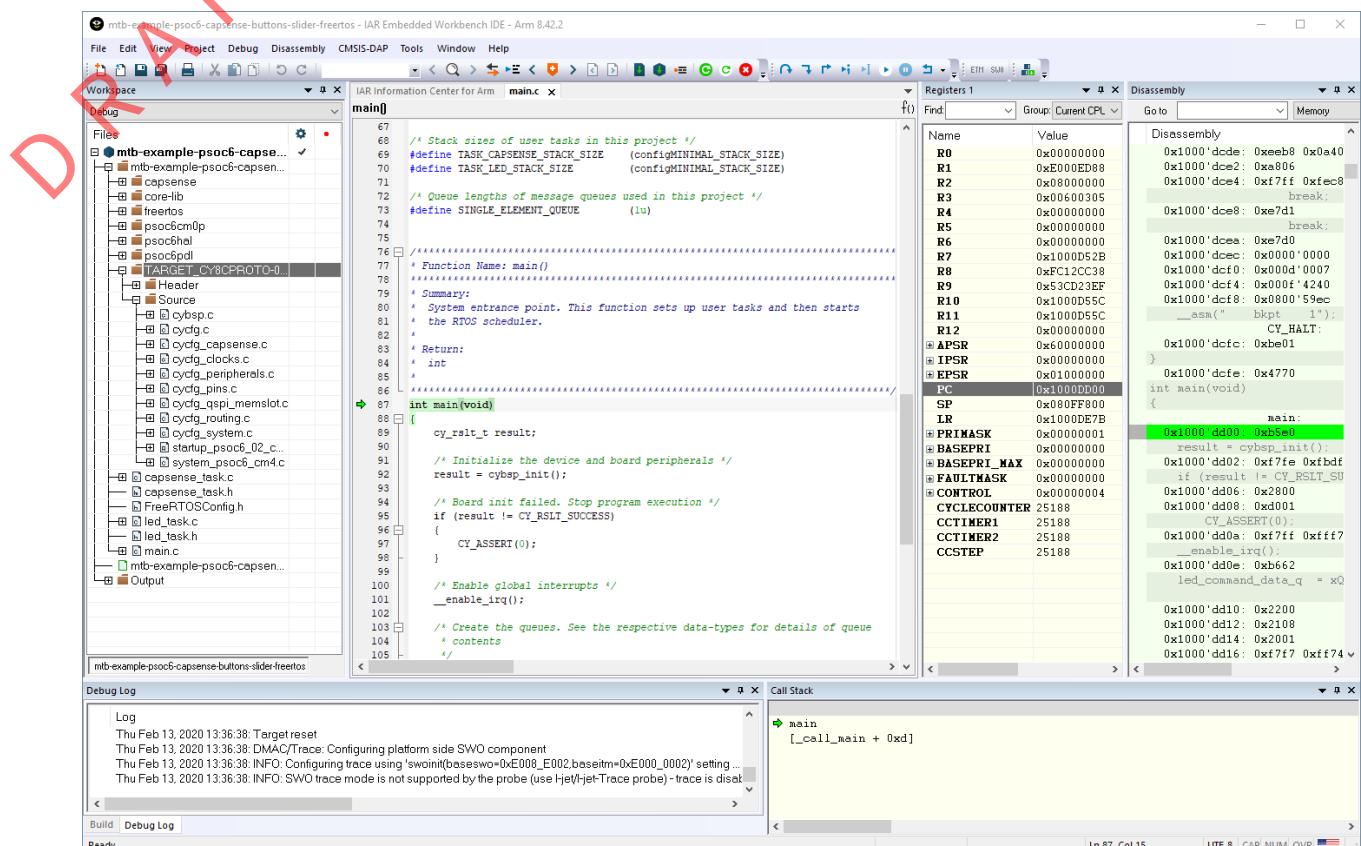
3. Select the **CMSIS-DAP** node, switch the interface from **JTAG** to **SWD**, and set the Interface speed to **2MHz**.



4. Click **OK**.
5. Select **Project > Download and Debug**.

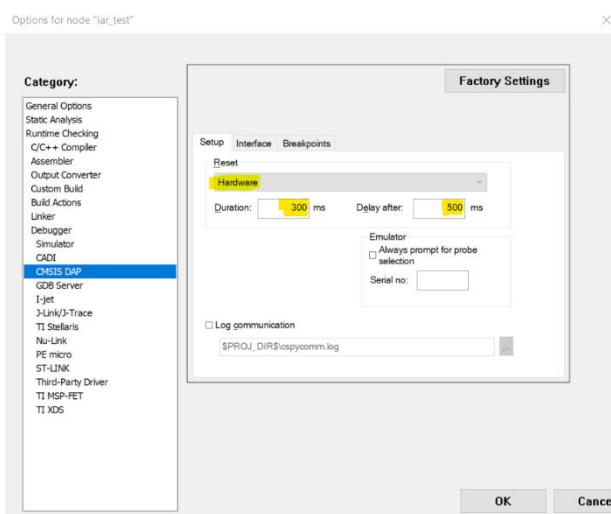
The IAR Embedded Workbench starts a debugging session and jumps to the main function.

3 PSoC™ 6 software tools



To use MiniProg4 with PSoC™ 6 single core and PSoC™ 6 256K

For a single-core PSoC™ 6 MCU, you must specify a special type of reset, as follows:



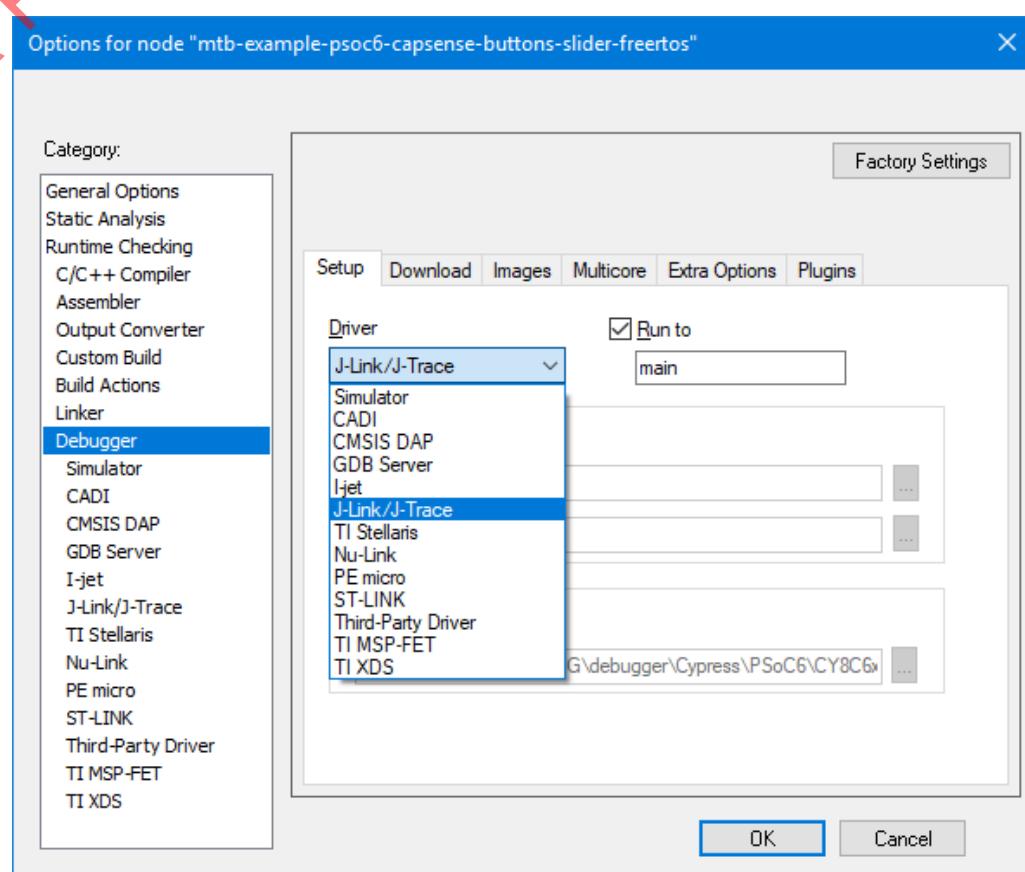
To use J-Link

You can use a J-Link debugger probe to debug the application.

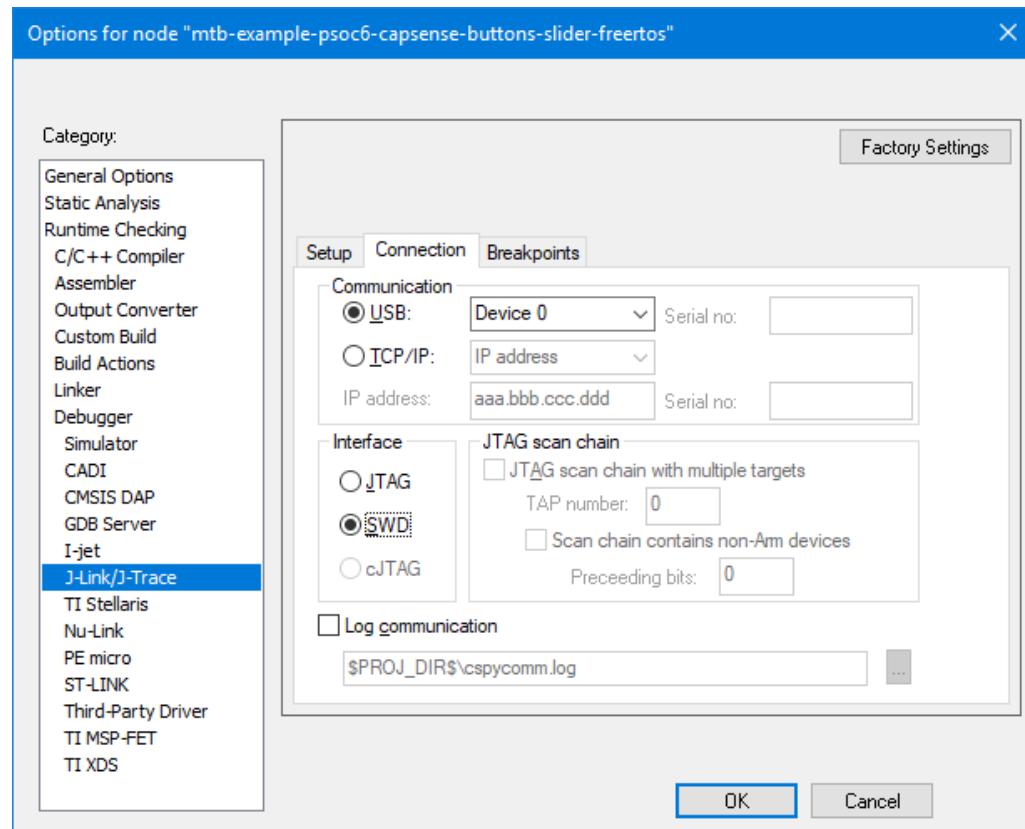
1. Open the Options dialog and select the **Debugger** item under **Category**.
2. Then select **J-Link/J-Trace** as the active driver:

3 PSoC™ 6 software tools

DRAFT

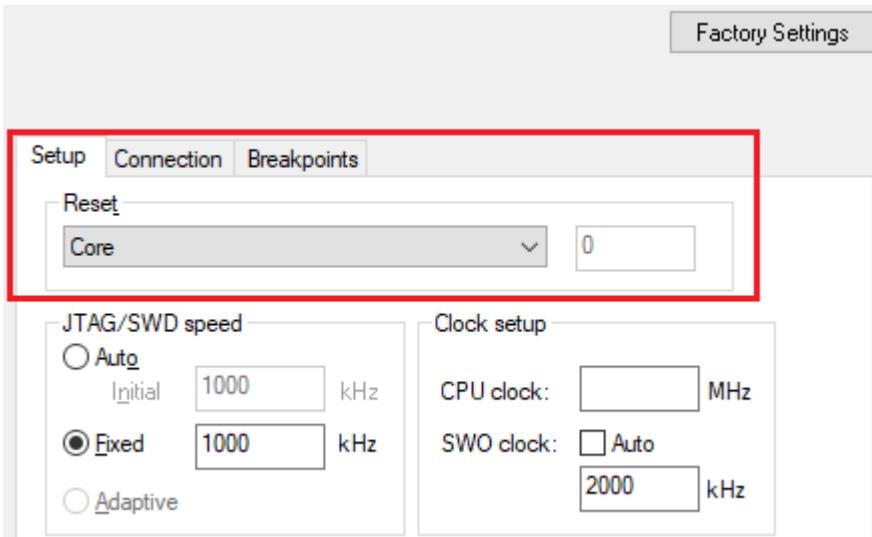


3. Select the **J-Link/J-Trace** item under **Category**, and under the **Connection** tab, switch the interface to **SWD**:



3 PSoC™ 6 software tools

Note: For PSoC™ 64 "Secure Boot" MCU, you must specify a special type of reset, as follows:



4. Connect a J-Link debug probe to the 10-pin adapter (needs to be soldered on the prototyping kits), and start debugging.

3.3.6.2.4 Export to Keil µVision 5 (Windows only)

This section describes how to export ModusToolbox™ application to Keil µVision and debug it with CMSIS-DAP or J-Link.

Prerequisites

- ModusToolbox™ 2.4 software and application
- Python 3.7 is installed in the tools_2.4 directory, and the make build system has been configured to use it. You don't need to do anything if you use the modus-shell/Cygwin.bat file to run command line tools. However, if you plan to use your own version of Cygwin or some other type of bash, you will need to ensure your system is configured correctly to use Python 3.7. Use the CY_PYTHON_PATH as appropriate.
- Keil µVision version 5.28 or later
- PSoC™ 6 Kit (for example, CY8CPROTO-062-4343W) with KitProg3 Firmware
- For J-Link debugging, download and install J-Link software:
https://www.segger.com/downloads/jlink/JLink_Windows.exe

Process example

1. Create a ModusToolbox™ application.
 - a. If you use the Project Creator tool, choose "ARM MDK" from the **Target IDE** pull down menu.
 - b. If you use the command line, open an appropriate shell program (see [CLI set-up instructions](#)), navigate to the application directory, and Run the following command:

```
make uvision5 TOOLCHAIN=ARM
```

Note: This sets the TOOLCHAIN to ARM in the Keil µVision configuration files but not in the ModusToolbox™ application's Makefile. Therefore, builds inside Keil µVision will use the ARM toolchain while builds from the ModusToolbox™ environment will continue to use the toolchain

3 PSoC™ 6 software tools

that was previously specified in the Makefile. You can edit the Makefile's TOOLCHAIN variable if you also want ModusToolbox™ builds to use the ARM toolchain.

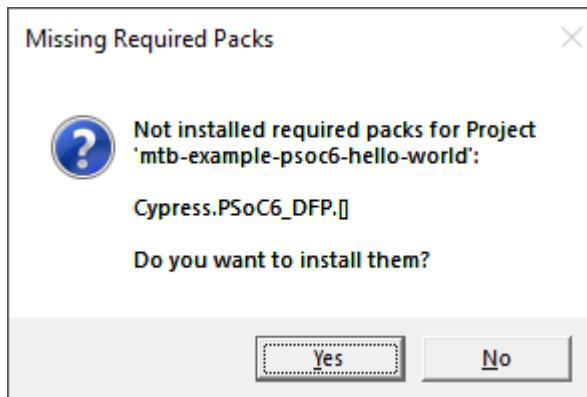
Note: Check the output log for instructions and information about various flags.

This generates the following files in the application directory:

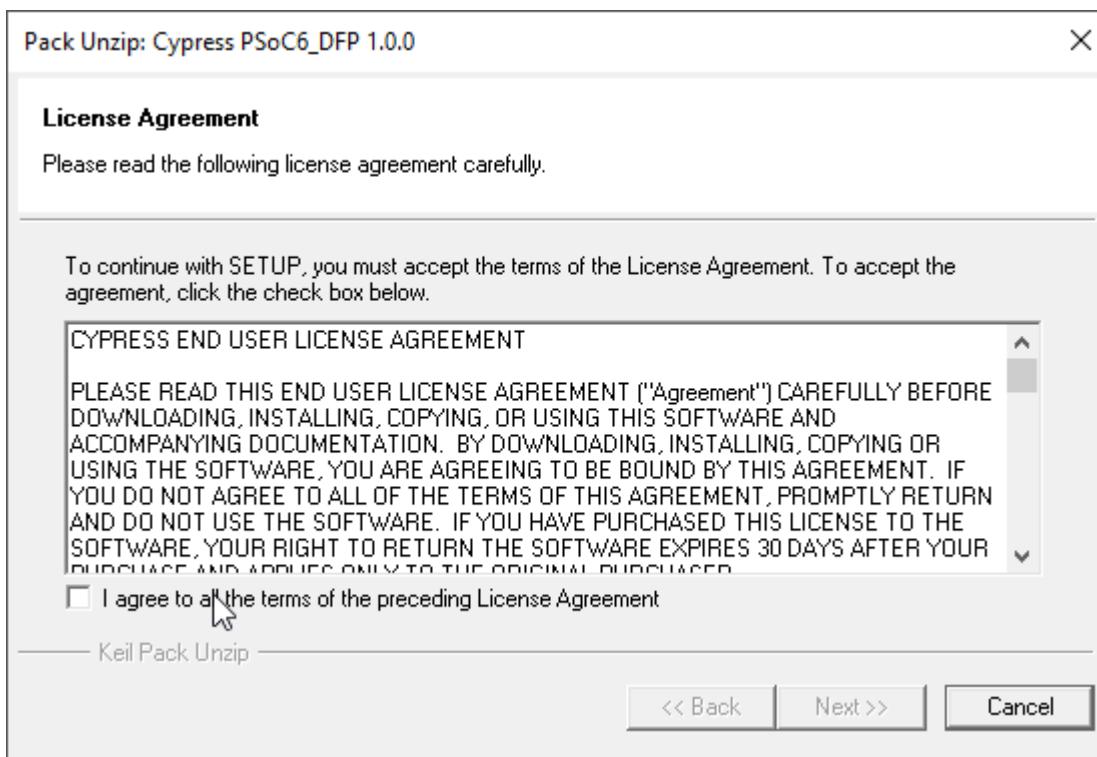
- mtb-example-psoc6-hello-world.cpdsc
- mtb-example-psoc6-hello-world.cprj
- mtb-example-psoc6-hello-world.gpdsc

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

2. Double-click the mtb-example-psoc6-hello-world file (either *.cpdsc or *.cprj, depending on version). This launches the Keil µVision IDE. The first time you do this, the following dialog displays:

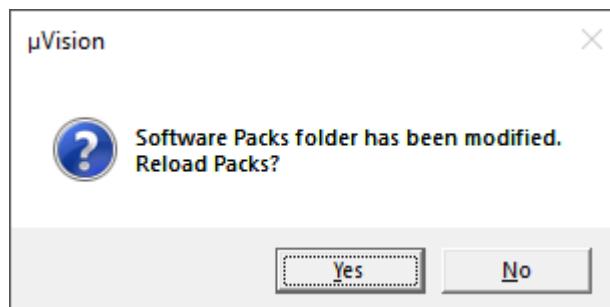


3. Click **Yes** to install the device pack. You only need to do this once.
4. Follow the steps in the Pack Installer to properly install the device pack.



3 PSoC™ 6 software tools

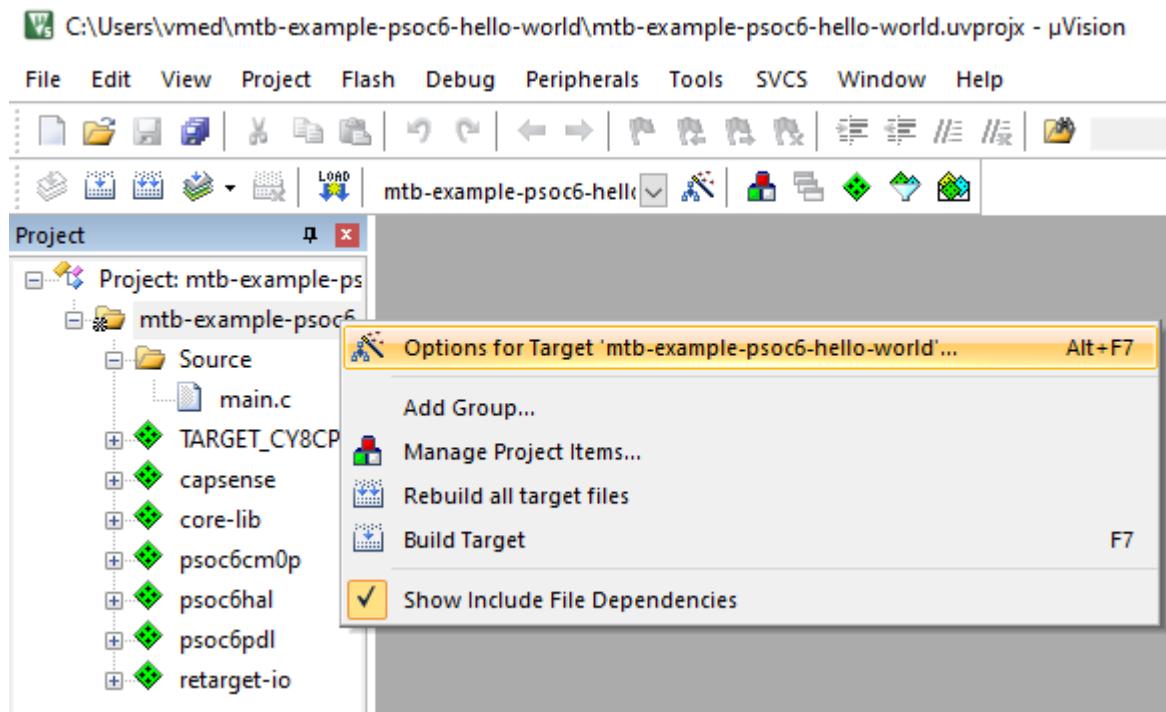
DRAFT



Note: In some cases, you may see the following error message:SSL caching disabled in Windows Internet settings. Switched to offline mode. See this link for how to solve this problem: <https://developer.arm.com/documentation/ka002253/latest>

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

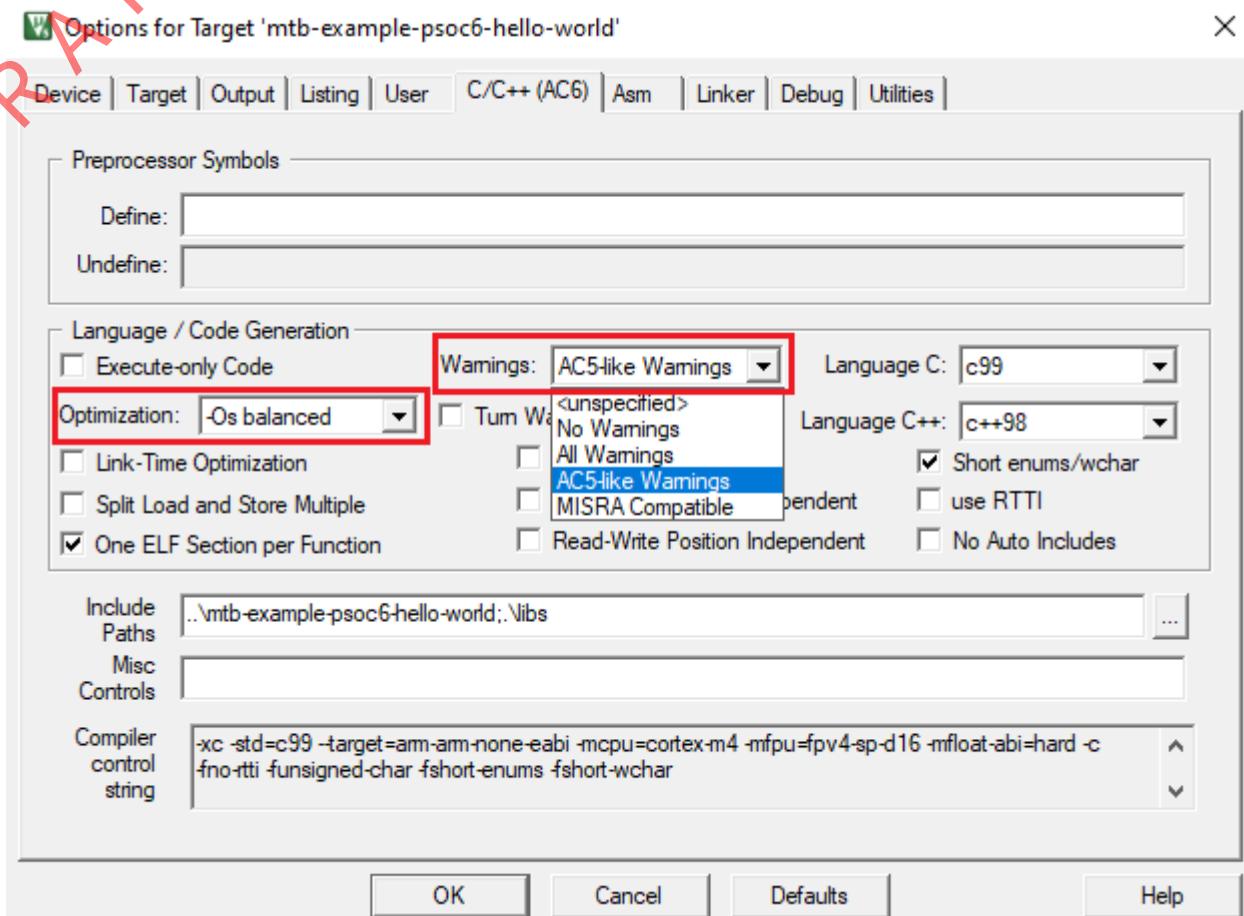
5. Right-click on the mtb-example-psoc6-hello-world directory in the µVision Project view, and select **Options for Target '<application-name>' ...**



6. On the dialog, select the **C/C++ (AC6)** tab.
 - Check that the Language C version was automatically set to c99.
 - Select "AC5-like warnings" in the Warnings drop-down list.
 - Select "-Os balanced" in the Optimization drop-down list.

3 PSoC™ 6 software tools

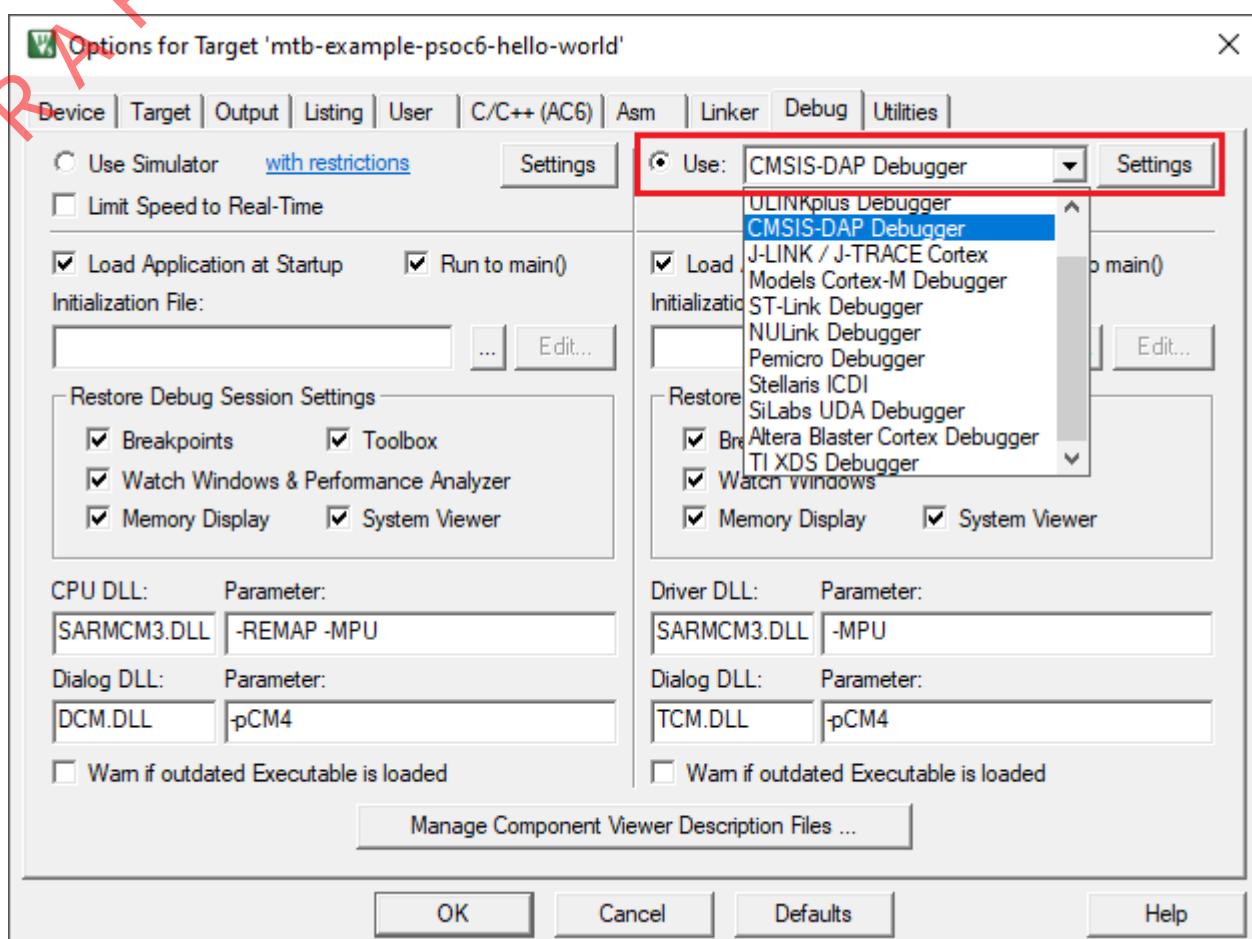
DRAFT



7. Select the **Debug** tab, and select KitProg3 CMSIS-DAP as an active debug adapter:

3 PSoC™ 6 software tools

DRAFT



- Click **OK** to close the Options dialog.

- Select **Project > Build target**.

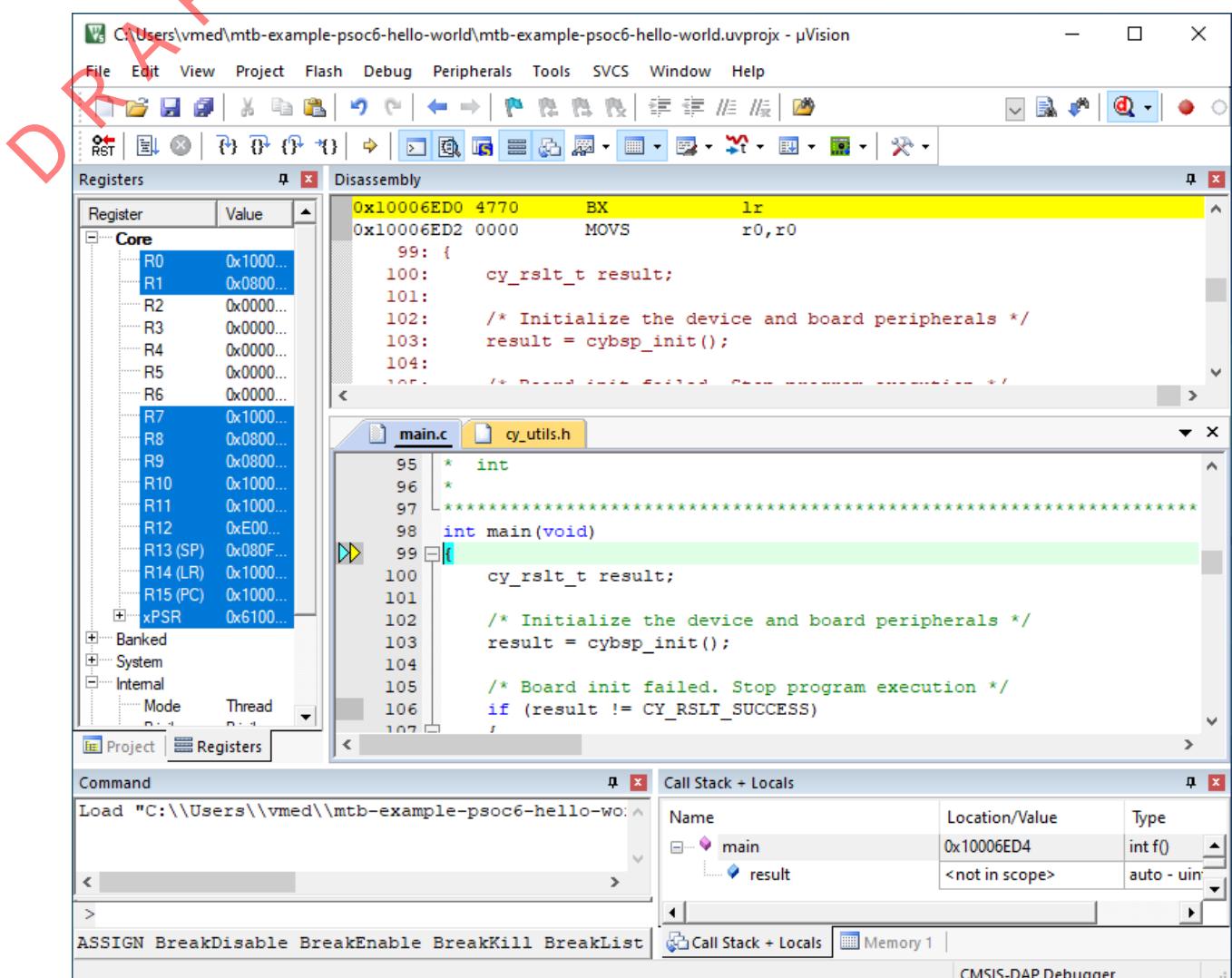
```
Build Output
compiling cy_retarget_io.c...
linking...
.libs\TARGET_CY8CPROTO-062-4343W\COMPONENT_CM4\TOOLCHAIN_ARM\cy8c6xxa_cm4_dual.sct(144): warning: L6329W: Pattern *(.cy_ramfunc) only matches removed unused sections.
.libs\TARGET_CY8CPROTO-062-4343W\COMPONENT_CM4\TOOLCHAIN_ARM\cy8c6xxa_cm4_dual.sct(170): warning: L6314W: No section matches pattern *(.cy_app_signature).
.libs\TARGET_CY8CPROTO-062-4343W\COMPONENT_CM4\TOOLCHAIN_ARM\cy8c6xxa_cm4_dual.sct(180): warning: L6314W: No section matches pattern *(.cy_em_eeprom).
.libs\TARGET_CY8CPROTO-062-4343W\COMPONENT_CM4\TOOLCHAIN_ARM\cy8c6xxa_cm4_dual.sct(189): warning: L6314W: No section matches pattern *(.cy_sflash_user_data).
.libs\TARGET_CY8CPROTO-062-4343W\COMPONENT_CM4\TOOLCHAIN_ARM\cy8c6xxa_cm4_dual.sct(198): warning: L6314W: No section matches pattern *(.cy_sflash_nar).
.libs\TARGET_CY8CPROTO-062-4343W\COMPONENT_CM4\TOOLCHAIN_ARM\cy8c6xxa_cm4_dual.sct(207): warning: L6314W: No section matches pattern *(.cy_sflash_public_key).
.libs\TARGET_CY8CPROTO-062-4343W\COMPONENT_CM4\TOOLCHAIN_ARM\cy8c6xxa_cm4_dual.sct(216): warning: L6314W: No section matches pattern *(.cy_toc_part2).
.libs\TARGET_CY8CPROTO-062-4343W\COMPONENT_CM4\TOOLCHAIN_ARM\cy8c6xxa_cm4_dual.sct(225): warning: L6314W: No section matches pattern *(.cy_rtoc_part2).
.libs\TARGET_CY8CPROTO-062-4343W\COMPONENT_CM4\TOOLCHAIN_ARM\cy8c6xxa_cm4_dual.sct(235): warning: L6314W: No section matches pattern *(.cy_xip).
.libs\TARGET_CY8CPROTO-062-4343W\COMPONENT_CM4\TOOLCHAIN_ARM\cy8c6xxa_cm4_dual.sct(245): warning: L6314W: No section matches pattern *(.cy_efuse).
.libs\TARGET_CY8CPROTO-062-4343W\COMPONENT_CM4\TOOLCHAIN_ARM\cy8c6xxa_cm4_dual.sct(253): warning: L6314W: No section matches pattern *(.cymeta).

Program Size: Code=19998 RO=data=8386 RW=data=440 ZI=data=103796
Finished: 0 information, 11 warning and 0 error messages.
"..\mtb-example-psoc6-hello-world\build\mtb-example-psoc6-hello-world.axf" - 0 Error(s), 11 Warning(s).
Build Time Elapsed: 00:01:31
```

To suppress the linker warnings about unused sections defined in the linker scripts, add "6314,6329" to the Disable Warnings setting in the Project Linker Options.

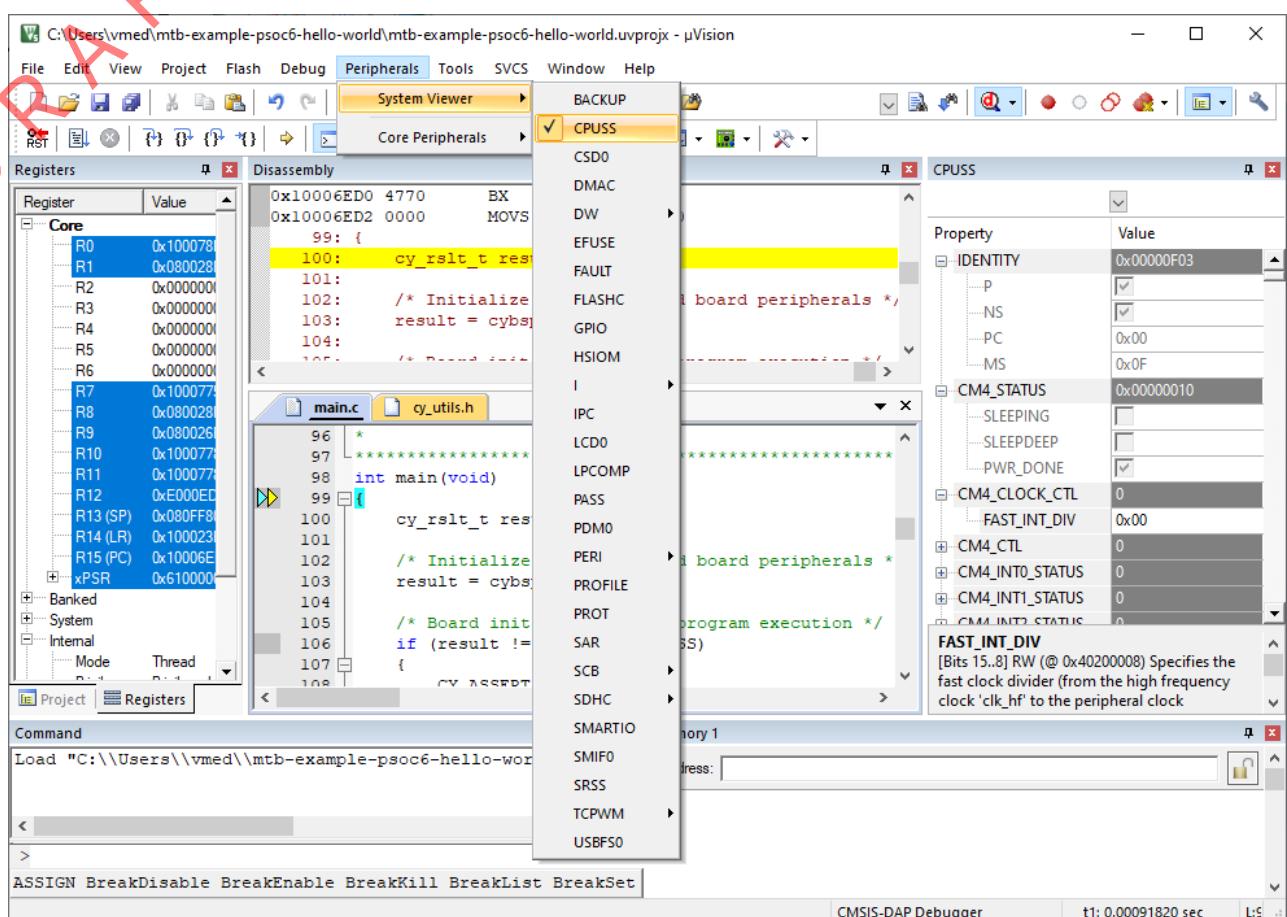
- Connect the PSoC™ 6 kit to the host PC.
- As needed, run the fw-loader tool to make sure the board firmware is upgraded to KitProg3. See [KitProg3 User Guide](#) for details. The tool is located in this directory by default:
<user_home>/ModusToolbox/tools_2.4/fw-loader/bin/
- Select **Debug > Start/Stop Debug Session**.

3 PSoC™ 6 software tools



You can view the system and peripheral registers in the SVD view.

3 PSoC™ 6 software tools

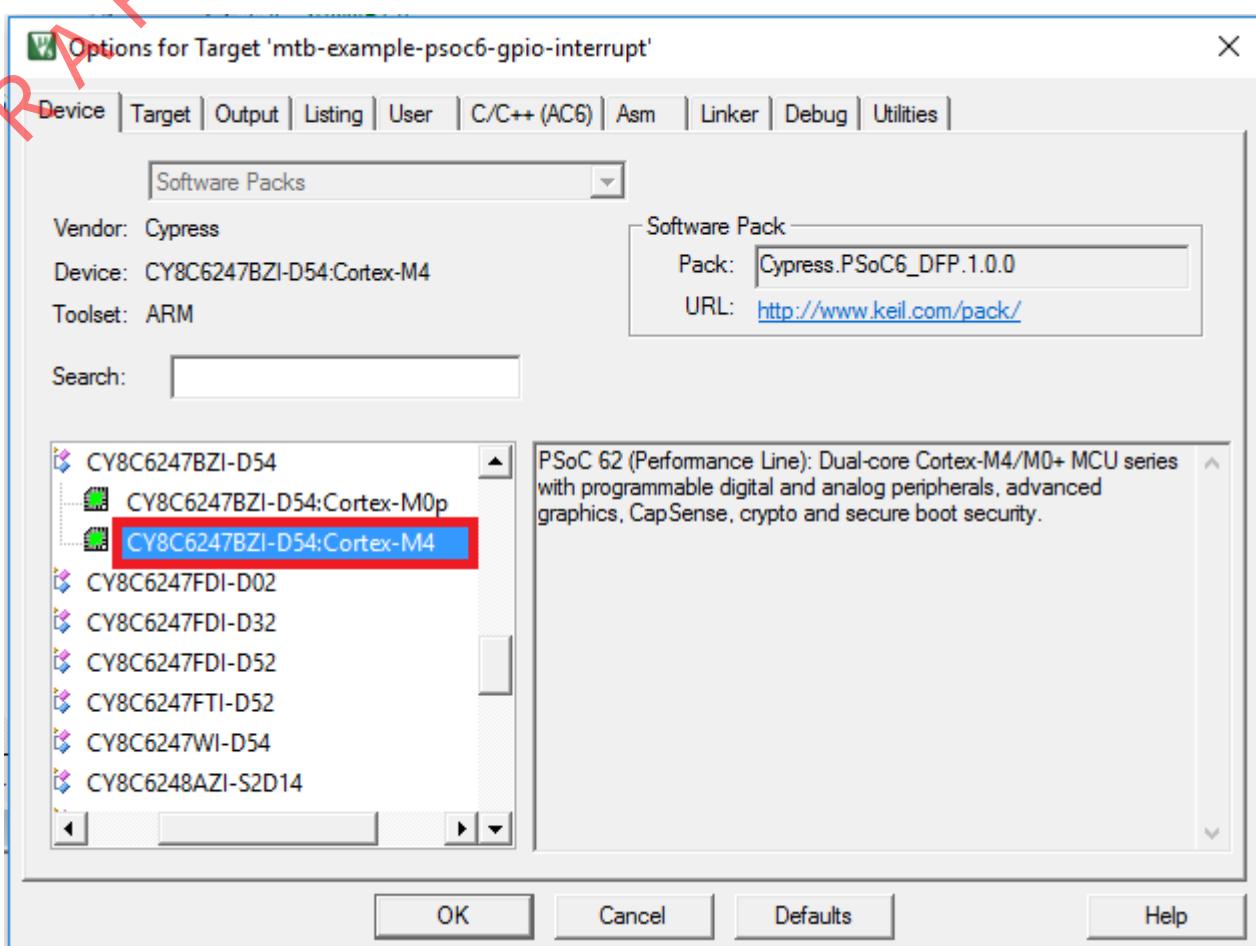


To use KitProg3/MiniProg4, CMSIS-DAP, and ULink2 debuggers

1. Select the **Device** tab in the Options for Target dialog and check that M4 core is selected:

3 PSoC™ 6 software tools

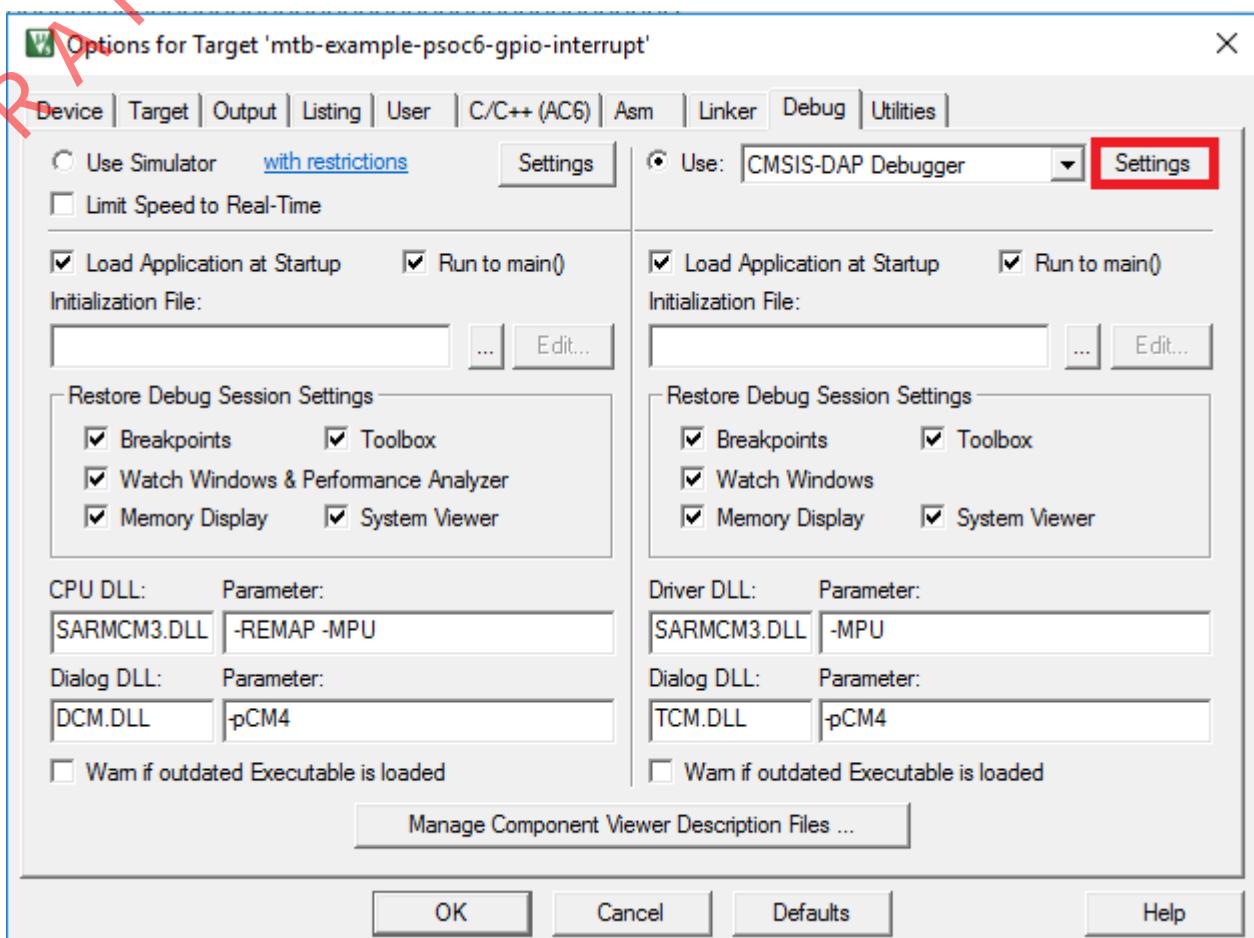
DRAFT



2. Select the **Debug** tab and click "Settings" to display the dialog **Target Driver Setup**:

3 PSoC™ 6 software tools

DRAFT

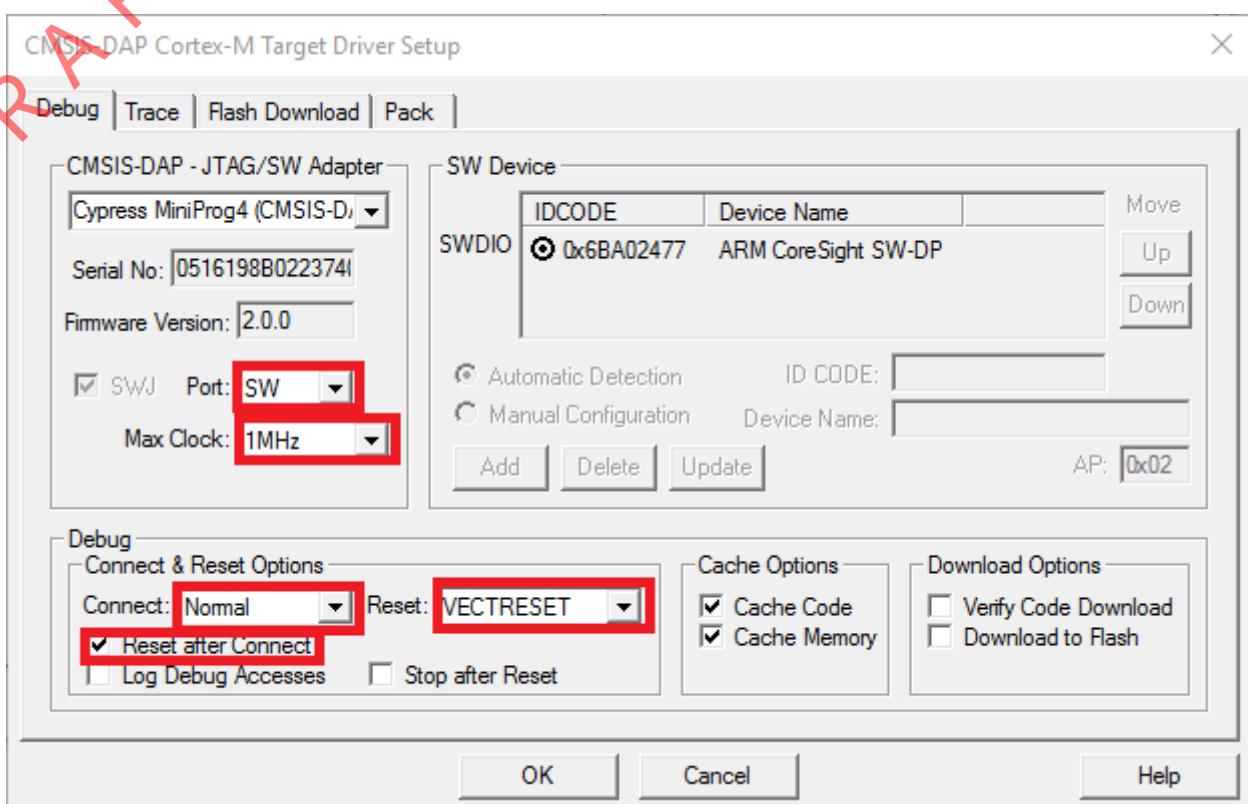


3. On the Target Driver Setup dialog, on the **Debug** tab, select the following:

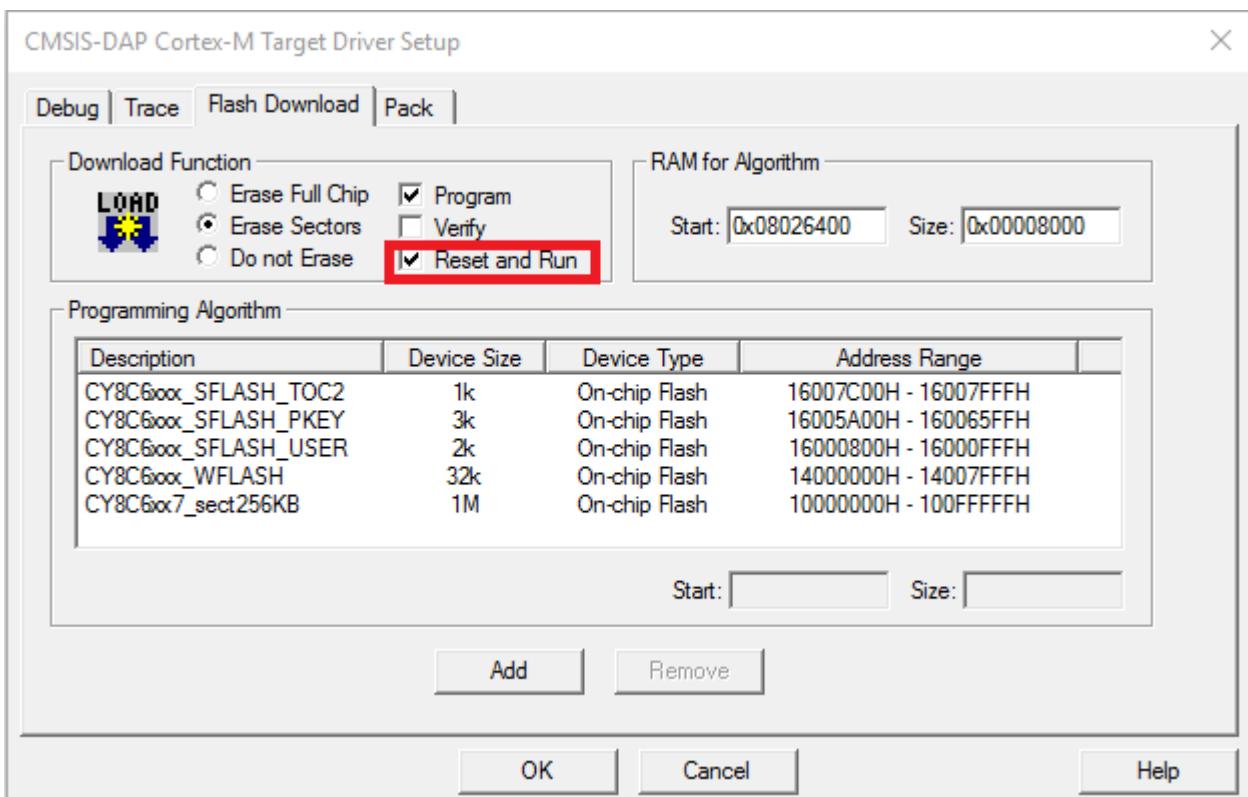
- set **Port** to "SW"
- set **Max Clock** to "1 MHz"
- set **Connect** to "Normal"
- set **Reset**:
 - For PSoC™ 6, to "VECTRESET"
 - For PSoC™ 4, PMG1, and AIROC™ CYW208xx, to "SYSRESETREQ"
- enable **Reset after Connect** option

3 PSoC™ 6 software tools

DRAFT



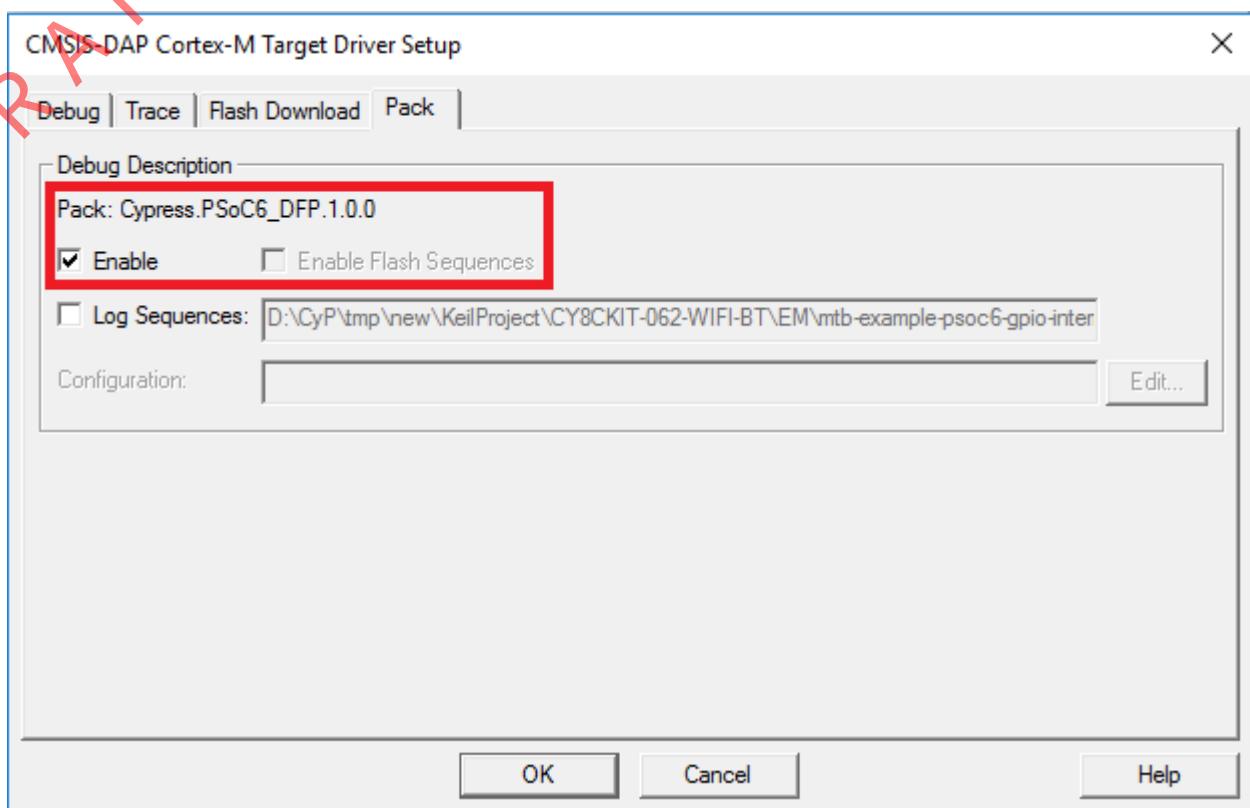
4. Select the **Flash Download** tab and select "Reset and Run" option after download, if needed:



5. Select the **Pack** tab and check if "Cypress.PSoC6_DFP" is enabled:

3 PSoC™ 6 software tools

DRAFT

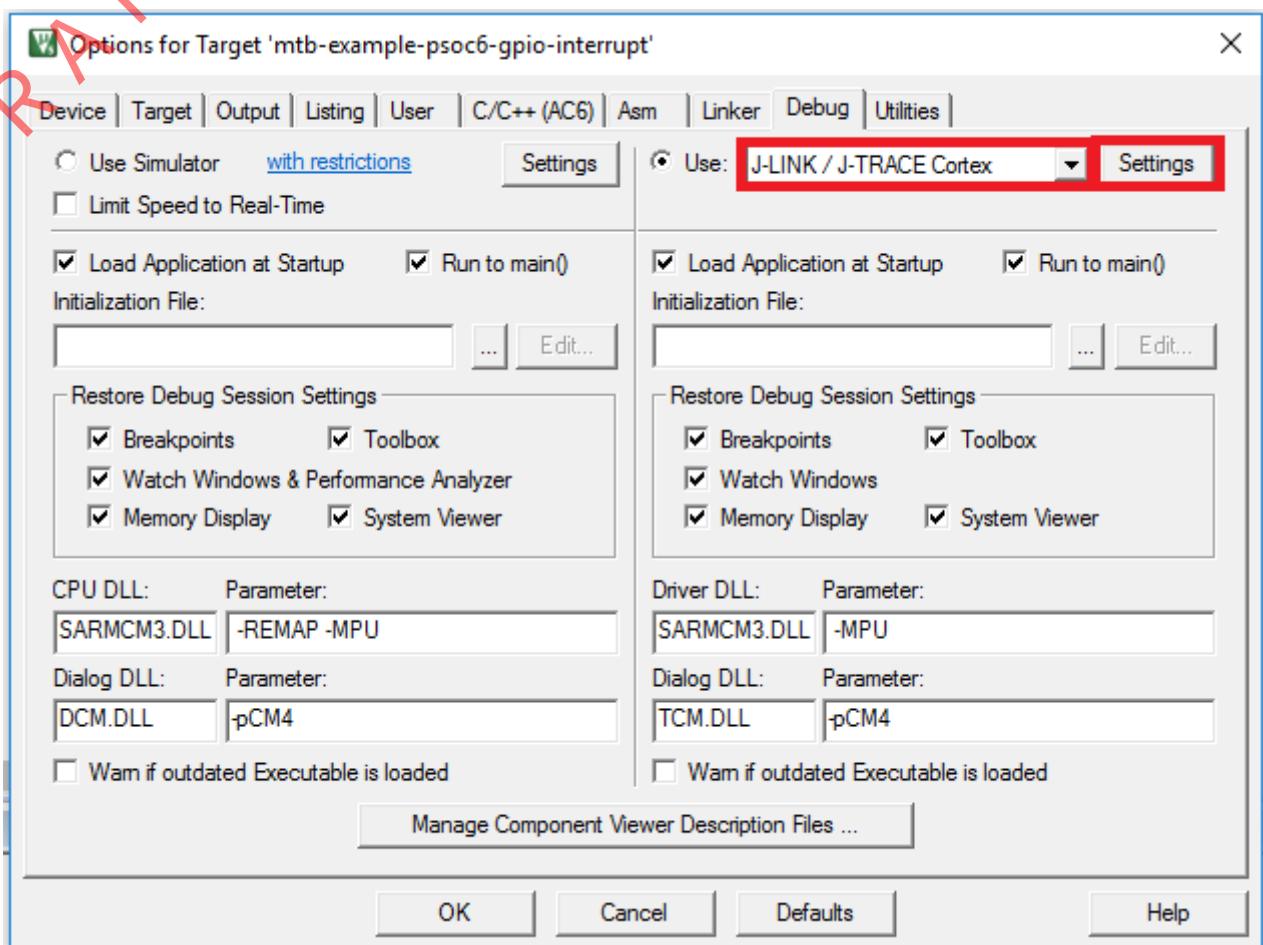


To use J-Link debugger

1. Make sure you have J-Link software version 6.62 or newer.
2. Select the **Debug** tab in the Options for Target dialog, select **J-LINK / J-TRACE Cortex** as debug adapter, and click **Settings**:

3 PSoC™ 6 software tools

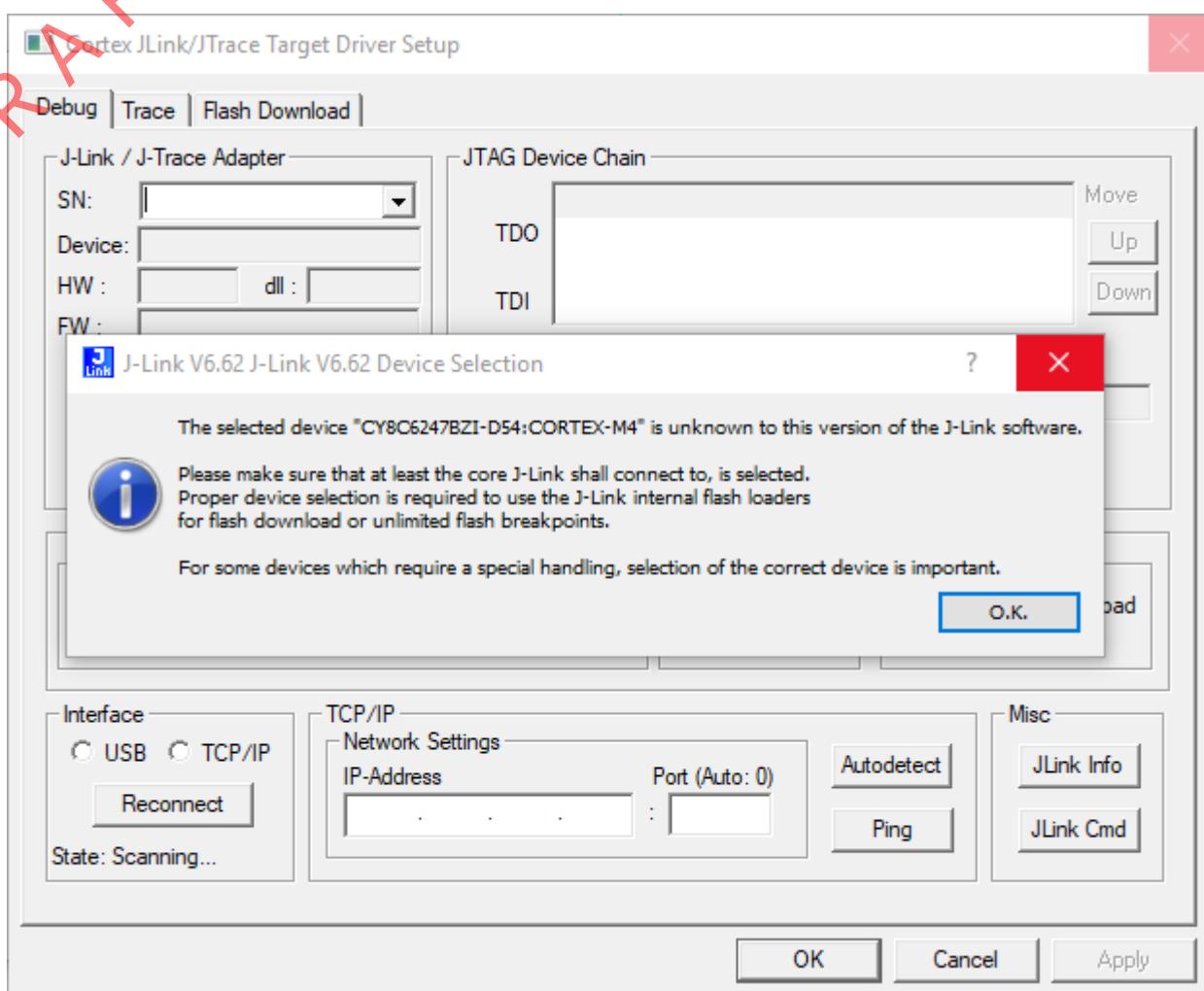
DRAFT



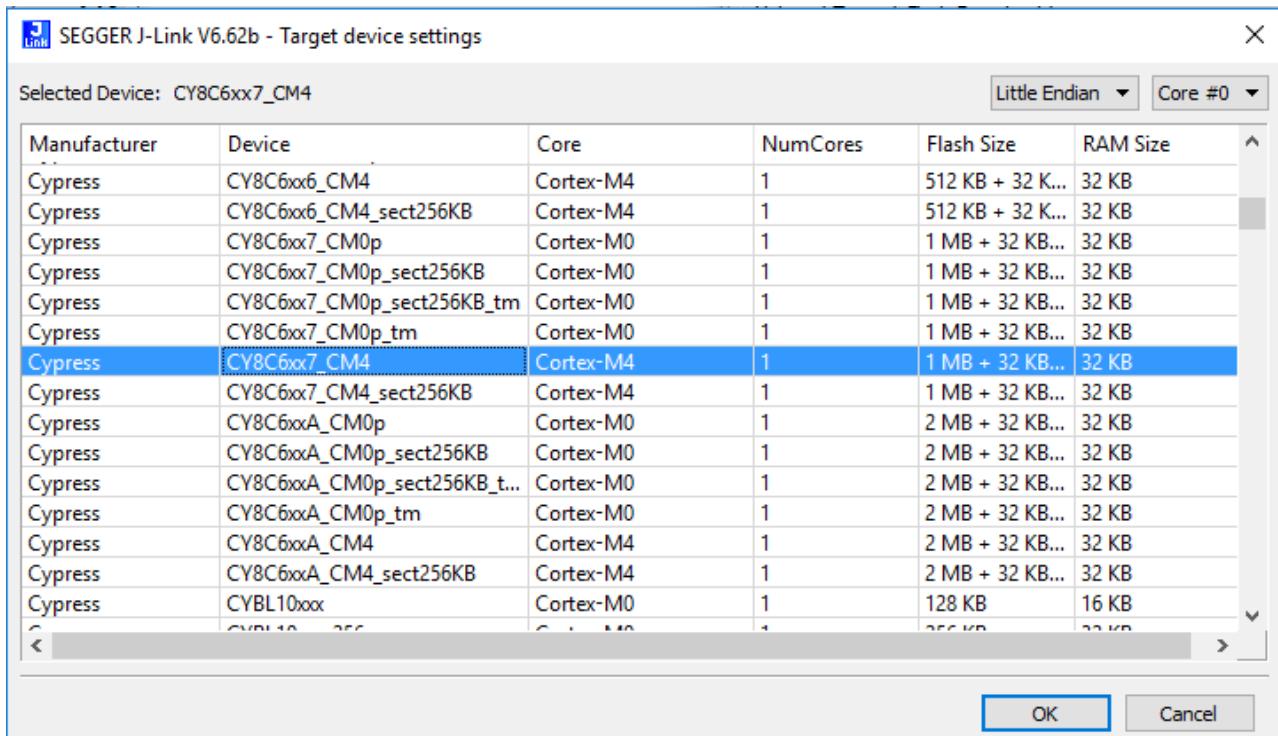
3. Click **OK** in the Device selection message box:

3 PSoC™ 6 software tools

DRAFT



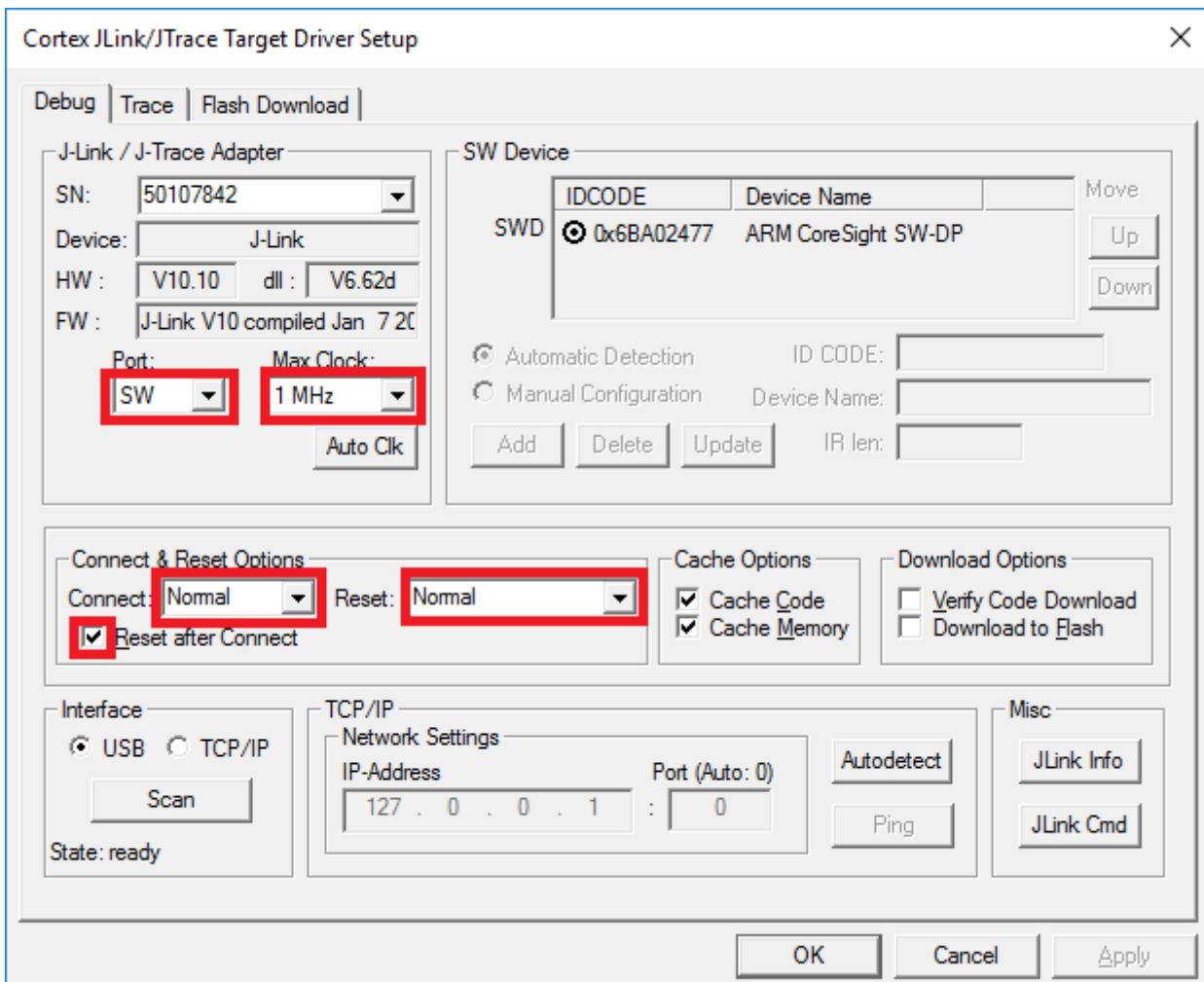
4. Select appropriate target in Wizard:



~~3 PSoC™ 6 software tools~~

5. Go to **Debug** tab in **Target Driver Setup** dialog and select:

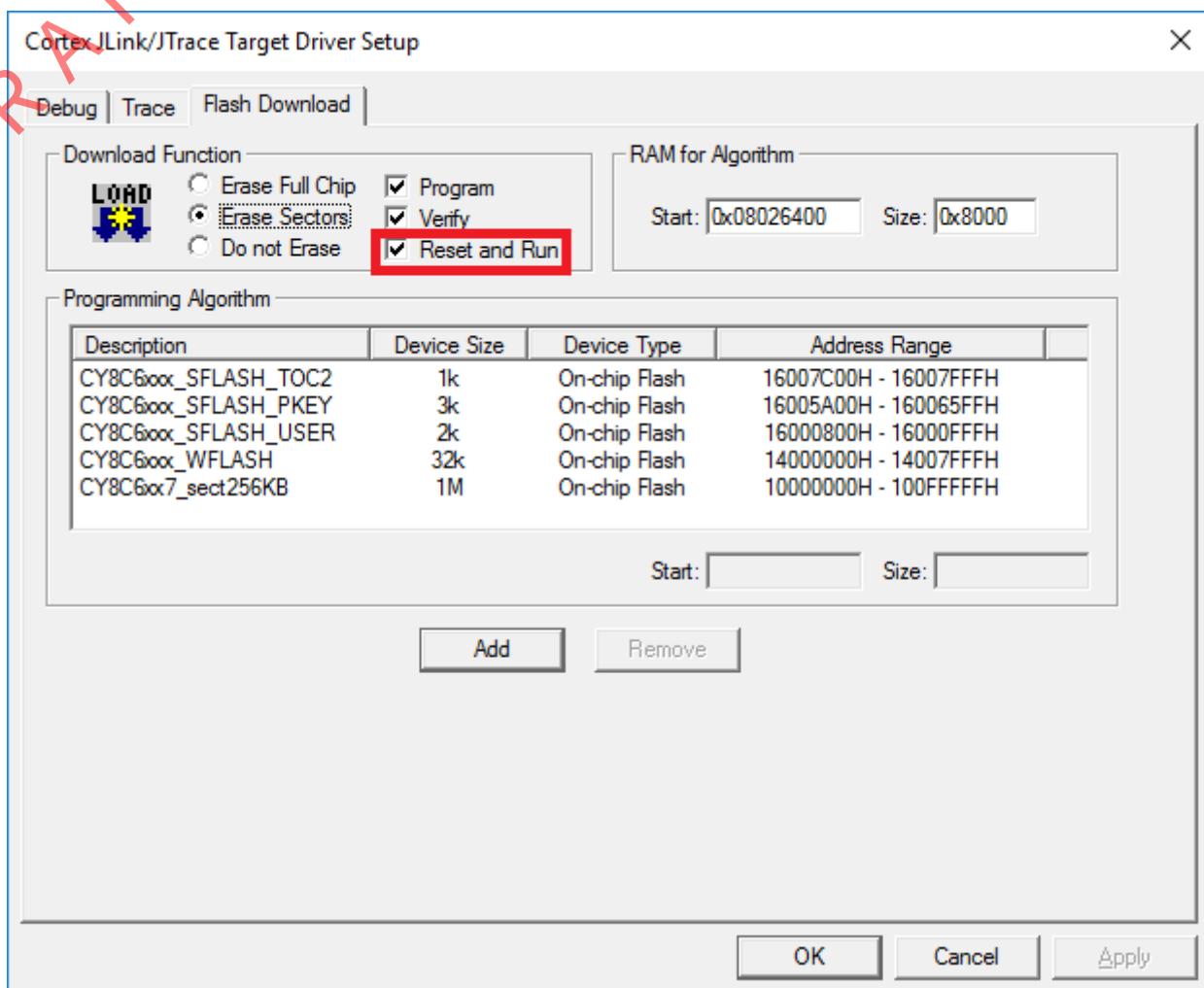
- set **Port** to "SW"
- set **Max Clock** to "1 MHz"
- set **Connect** to "Normal"
- set **Reset** to "Normal"
- enable **Reset after Connect** option



6. Select the **Flash Download** tab in the Target Driver Setup dialog and select **Reset and Run** option after download if needed:

3 PSoC™ 6 software tools

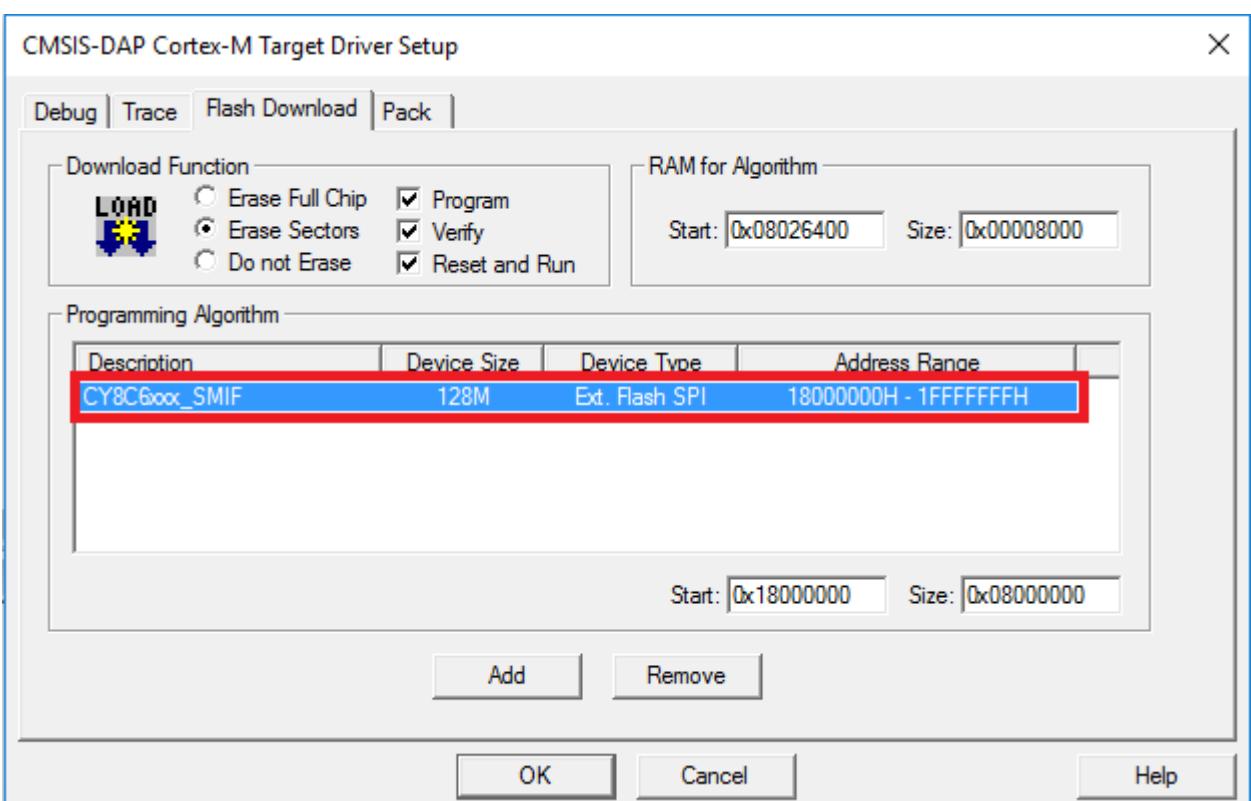
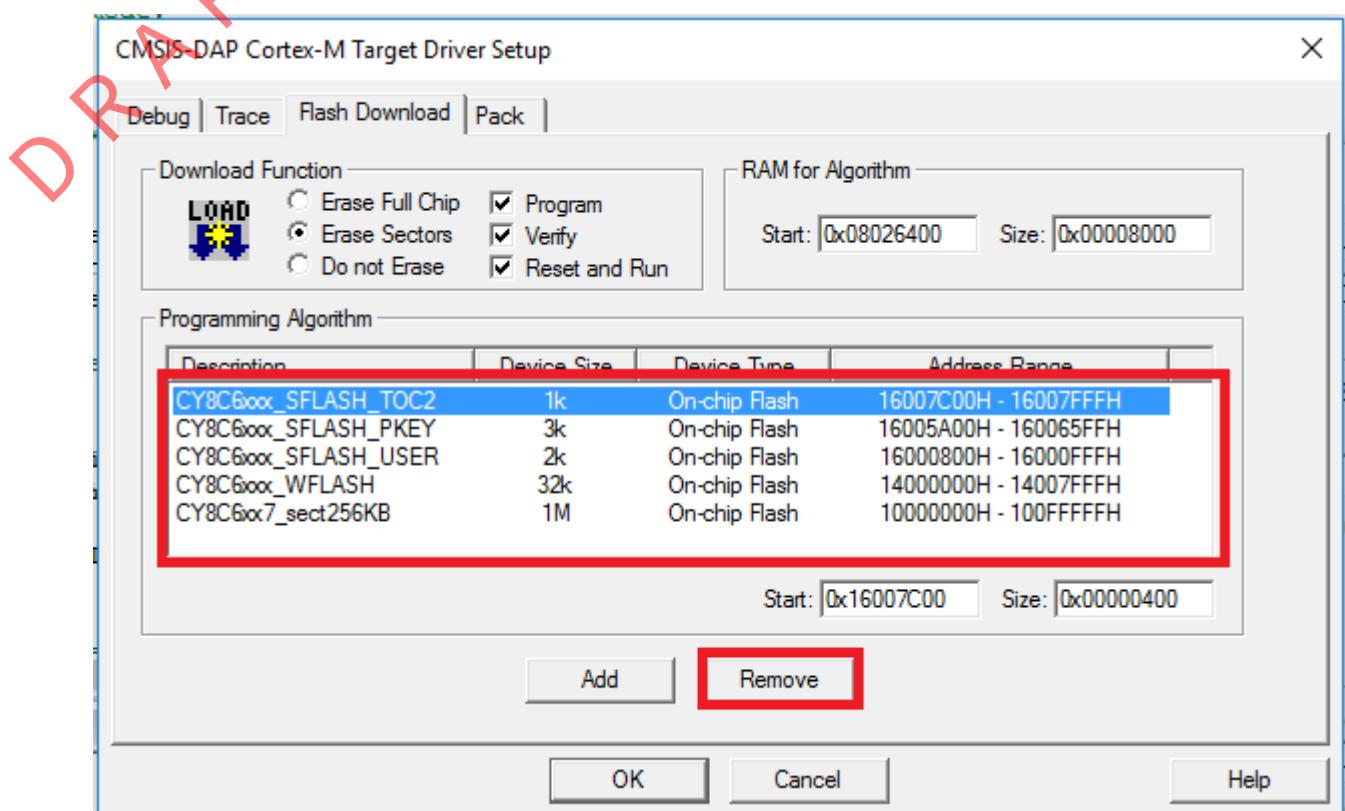
DRAFT



Program external memory

1. Download internal flash as described above.
Notice "No Algorithm found for: 18000000H - 1800FFFFH" warning.
2. Select the **Flash Download** tab in the Target Driver Setup dialog and remove all programming algorithms for On-chip Flash and add programming algorithm for External Flash SPI:

3 PSoC™ 6 software tools



3. Download flash.

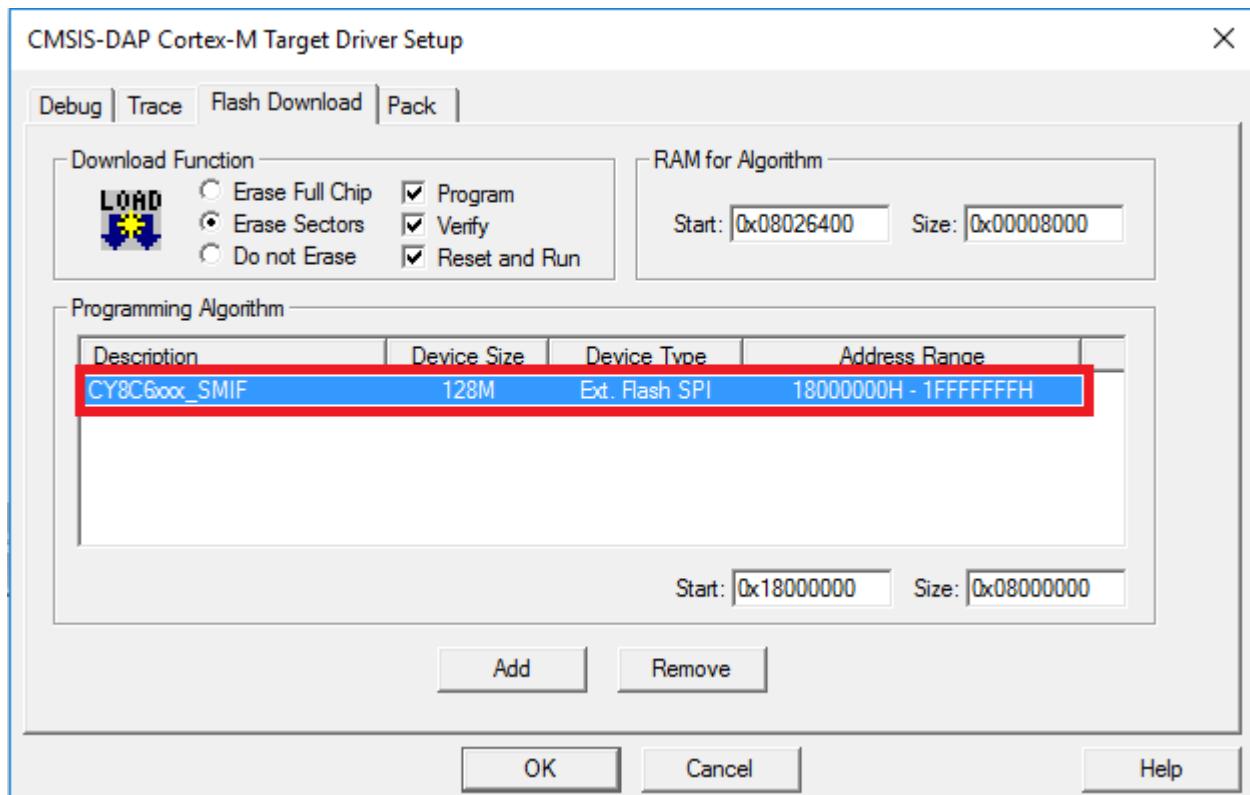
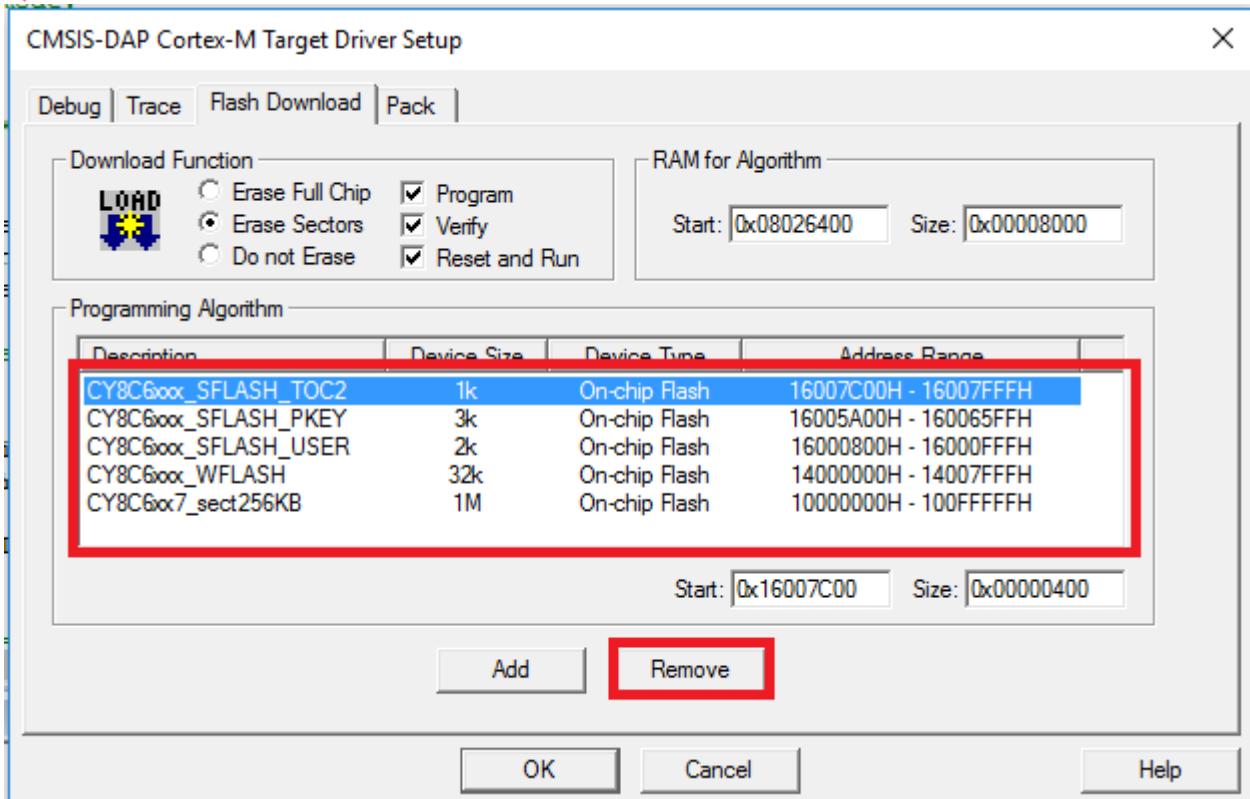
Notice warnings:

- No Algorithm found for: 10000000H - 1000182FH
- No Algorithm found for: 10002000H - 10007E5BH
- No Algorithm found for: 16007C00H - 16007DFFH

~~3 PSoC™ 6 software tools~~

~~Erase external memory~~

- Select the **Flash Download tab** in Target Driver Setup dialog and remove all programming algorithms for On-chip Flash and add programming algorithm for External Flash SPI:



- Click **Flash > Erase** in menu bar.

~~3 PSoC™ 6 software tools~~

~~3.3.6.3 Patched flashloaders for AIROC™ CYW208xx devices~~

To enable support for different QSPI settings, the ModusToolbox™ QSPI Configurator patches flashloaders and stores FLM files for them in the application directory. When exporting such applications to 3rd party IDEs (for example, Keil µVision or IAR EWARM), these patched flashloader files must be copied into the appropriate 3rd party IDE directory.

1. Copy the flashloader file located in the <app-dir>\libs\<Kit-Name>\COMPONENT_BSP_DESIGN_MODUS\GeneratedSource directory.
 - For Keil µVision, copy the CYW208xx_SMIF.FLM file.
 - For IAR EWARM, copy the CYW208xx_SMIF.out file.
2. Paste the flashloader file as follows:
 - For Keil µVision, paste to the C:\Users\<User-Name>\AppData\Local\Arm\Packs\Cypress\CYW208xx_DFP\<Version>\Flash directory.
 - For IAR EWARM, paste to the C:\Program Files\IAR Systems\Embedded Workbench 9.0\arm\config\flashloader\Infineon\CYW208XX directory.
3. Also, to use the SEGGER J-Link debugger, paste the CYW208xx_SMIF.FLM file to the C:\Program Files\SEGGER\JLink\Devices\Cypress\cat1b directory.

3.3.6.4 Generating files for XMC™ Simulator tool

For the XMC1100, XMC1200, XMC1300, and XMC1400 families of devices, you can generate an archive file to upload to the XMC™ Simulator tool (<https://design.infineon.com/tinaui/designer.php>) for simulation and debugging. To do this:

Specify the CY_SIMULATOR_GEN_AUTO=1 variable as follows:

- Edit the application Makefile to add the CY_SIMULATOR_GEN_AUTO=1 variable, and then build the application, or
- Add the variable on the command line: `make build CY_SIMULATOR_GEN_AUTO=1`

When the build completes, it generates an archive file (<application-name>.tar.tgz) in the <Application-Name>\build\<Kit-Name>\Debug directory, and the build message displays the URL to the appropriate simulator tool. For example:

```
=====
= Generating simulator archive file =
=====
The Infineon online simulator link:
https://design.infineon.com/tinaui/designer.php?path=EXAMPLESROOT%7CINFINEON%7CApplications%7CIndustrial%7C&file=mcu\_XMC1200\_Boot\_Kit\_MTB\_v2.tsc
Simulator archive file C:/Users/XYZ/mtw2.4/5699/xmc-2/Empty_XMC_App/
build/KIT_XMC12_BOOT_001/Debug/mtb-example-xmc-empty-app.tar.tgz successfully generated
```

- If using the Eclipse IDE, click the link in the Quick Panel under **Tools** to open the XMC™ Simulator tool in the default web browser.
- If building with the command line, open a web browser to the URL displayed in the output message.

Upload the generated archive file to the XMC™ Simulator tool, and follow the tool's instructions for using the tool as appropriate.

4 PSoC™ 6 code examples

Revision history

DRAFT

-
- 3
- 6

Date	Revision	Description of change
3/24/2020	**	New document.
3/27/2020	*A	Updates to screen captures and associated text.
4/1/2020	*B	Fix broken links.
4/29/2020	*C	Fix incorrect link.
8/28/2020	*D	Updates for ModusToolbox™ 2.2.
9/23/2020	*E	Corrections to Build system and Board support packages chapters.
9/29/2020	*F	Added links to KBAs; updated text for cyignore.
10/2/2020	*G	Added details for BTSDK v2.8 BSPs/libraries.
1/14/2021	*H	Updated Manifest chapter and fixed broken links.
3/23/2021	*I	Updates for ModusToolbox™ 2.3.
5/24/2021	*J	Updated information for creating a custom BSP.
9/27/2021	*K	Updates for ModusToolbox™ 2.4.
11/29/2021	*L	Merged chapter 3 (software overview) into chapter 1 (introduction). Updated sections 6.2.3 and 6.2.4 with notes and minor details. Added section 6.3 with information for patched flashloaders and 3 rd party IDEs.
2/24/2022	*M	Added link to PSoC™ 4 Application Note.
3/8/2022	*N	Updated for version 3.0. Updated Export instructions to include AIROC™ CYW208xx devices.

4 PSoC™ 6 code examples

Getting Started

Empty App	This empty application provides a template for creating applications. For more details, see the README on GitHub .
Hello World	This code example demonstrates simple UART communication by printing a "Hello World" message on a terminal and blinks an LED using a Timer resource. For more details, see the README on GitHub .
Dual-CPU Empty PSoC6 App	This empty application provides a template for creating dual-CPU applications using PSoC™ 6 devices. For more details, see the README on GitHub .
Switching Power Modes	This example demonstrates how to transition PSoC™ 6 between the following power modes - Active, Sleep, Low Power Active, Low Power Sleep, and Deep Sleep. For more details, see the README on GitHub .
Security App	This code example is a minimal starter dual-CPU security application template for PSoC™ 62/63 MCU devices that demonstrates secure boot, memory protection, protected storage, device firmware update and signing your application. For more details, see the README on GitHub .

~~4 PSoC™ 6 code examples~~

~~Peripherals~~

HAL Low-Power Timer	This example explains how to configure up a low-power timer using the LPTimer HAL resource to measure the timing between events in free-running mode. For more details, see the README on GitHub .
Fault Handling	This example demonstrates the fault handling functionality of PSoC™ 6 MCU using Peripheral Driver Library (PDL) System Library (SysLib). For more details, see the README on GitHub .
HAL Watchdog Timer	This example explains how to set up a Watchdog Timer (WDT) using the WDT HAL resource. The WDT resets the device if it is not serviced or "kicked" within the configured timeout interval. This helps in recovering the program from an unintended lock up. For more details, see the README on GitHub .
Free-Running Multi-Counter Watchdog Timer	This example explains how to set up a Multi-Counter Watchdog Timer (MCWDT) using the MCWDT PDL resource to measure the timing between events in free-running mode on PSoC™ 6 MCU. For more details, see the README on GitHub .
HAL PWM Square Wave	This code example generates a square wave using the PWM driver. An LED connected to the PWM output pin blinks at 2 Hz. For more details, see the README on GitHub .
UART Transmit and Receive	This example demonstrates the UART (HAL) transmit and receive operation. For more details, see the README on GitHub .
SCB UART Transmit and Receive using DMA	This example demonstrates the UART transmit and receive operation using DMA in PSoC™ 6 MCU. For more details, see the README on GitHub .
HAL I2C Master	This example demonstrates the use of I2C (HAL) resource in Master mode. For more details, see the README on GitHub .
I2C Master EzI2C Slave	This example demonstrates the use of I2C (HAL) resource in Master mode with an EzI2C slave. For more details, see the README on GitHub .
I2C Slave Using Callbacks	This example demonstrates the operation of the I2C (HAL) resource in Slave mode using callbacks. For more details, see the README on GitHub .
HAL SPI Master	This example demonstrates the use of SPI (HAL) resource in Master mode. For more details, see the README on GitHub .
SCB SPI Master DMA	This example demonstrates the use of PSoC™ 6 MCU Serial Communication Block (SCB) resource in SPI Master mode using DMA. For more details, see the README on GitHub .
Cryptography SHA Demonstration	This code example shows how to generate a 32-byte hash value or message digest for an arbitrary user input message with the SHA2 algorithm using the Cryptographic hardware block. For more details, see the README on GitHub .
CSDADC	This example demonstrates the usage of CSD analog-to-digital converter (ADC) in PSoC™ 6 MCU. CSDADC measures the external voltage and displays the conversion result the terminal application. For more details, see the README on GitHub .
PDM PCM Audio	This example demonstrates how to use the pulse-density modulation/pulse-code modulation (PDM/PCM) hardware block in PSoC™ 6 MCU with a digital microphone. For more details, see the README on GitHub .
I2S Audio	This example demonstrates how to use the I2S hardware block in PSoC™ 6 MCU to interface with an audio codec. For more details, see the README on GitHub .
Dual-CPU IPC Pipes	This example demonstrates how to use the inter-processor communication (IPC) driver to implement a message pipe in PSoC™ 6 MCU. The pipe is used to send messages between CPUs. For more details, see the README on GitHub .

4 PSoC™ 6 code examples

Dual-CPU IPC Semaphore	This example demonstrates how to use the inter-processor communication (IPC) driver to implement a semaphore in PSoC™ 6 MCU. The semaphore is used as a lock to control access to a resource shared by the CPUs and synchronize the initialization instructions. For more details, see the README on GitHub .
Low-Power Analog Front End using OpAmp and SAR ADC	This example demonstrates the low-power analog features of PSoC™ 6 MCU using an OpAmp and SAR ADC. PDL is used for the application firmware. This example is supported only for the devices like CY8C62x4 which have an OpAmp and SAR ADC capable of operating in System Deep Sleep mode. For more details, see the README on GitHub .
USB Mass Storage File System	This example demonstrates how to configure the USB block in a PSoC™ 6 MCU device as a Mass Storage (MSC) device and run a file system (FatFs) through an external memory (microSD). This example uses FreeRTOS. For more details, see the README on GitHub .
SGPIO Target Interface	This example uses a SPI resource and Smart I/O in PSoC™ 6 MCU to implement the SGPIO Target interface. For more details, see the README on GitHub .
QSPI F-RAM Access	This code example demonstrates interfacing PSoC™ 6 MCU with an external QSPI F-RAM memory and access it in Single, Dual, or Quad SPI mode using the Serial Memory Interface (SMIF) block. This example uses the PSoC™ 6 MCU standard QSPI HAL driver library to build the QSPI F-RAM access API. For more details, see the README on GitHub .
I2S Master using Smart IO and SPI	This example uses a SPI resource and Smart I/O in PSoC™ 6 MCU to implement the I2S Master interface (TX only). For more details, see the README on GitHub .
PDM to I2S Audio	This example demonstrates how to route Pulse-Density Modulation (PDM) audio data to the Inter-IC Sound (I2S) Interface in PSoC™ 6 MCU. For more details, see the README on GitHub .
USB Audio Recorder	This example demonstrates the use of PSoC™ 6 MCU to implement an audio recorder using the USB Audio Device Class. For more details, see the README on GitHub .
USB HID Mouse	This example demonstrates how to configure the USB block in a PSoC™ 6 MCU as a HID. The device enumerates as a 3-button mouse. For more details, see the README on GitHub .
USB HID Generic	This example demonstrates how to configure the USB block in a PSoC™ 6 MCU as a Human Interface Device (HID). The device enumerates as a Generic HID device. For more details, see the README on GitHub .
HAL GPIO Interrupt	This example demonstrates how to configure a GPIO to generate an interrupt. For more details, see the README on GitHub .
RTC Basics	This example demonstrates the usage of the real-time clock (RTC). For more details, see the README on GitHub .
ADC basic	This example demonstrates use of the ADC (Analog to Digital Converter) HAL driver to perform voltage measurements. For more details, see the README on GitHub .
USB Suspend and Resume	This code example demonstrates how PSoC™ 6 USB detects a suspend condition, enters a low-power state, and restores normal operation when USB activity is resumed. For more details, see the README on GitHub .
USB Audio Device FreeRTOS	This example demonstrates how to use PSoC™ 6 MCU to implement a USB Audio Device and HID Audio Playback Control that connects to the PC via the USB interface. The example also uses FreeRTOS. For more details, see the README on GitHub .

~~4 PSoC™ 6 code examples~~

Cryptography TRNG Demonstration	This code example demonstrates generating a One-Time Password (OTP) using the True Random Number generation feature of MCU cryptography block. For more details, see the README on GitHub .
Cryptography AES Demonstration	This code example encrypts and decrypts user input data using the AES algorithm using a 128-bit key. The encrypted and decrypted data are displayed on a UART terminal emulator. For more details, see the README on GitHub .
Protection Units FreeRTOS	This example demonstrates how to use the protection units to isolate the CM0+ CPU memory from CM4. For more details, see the README on GitHub .
Basic Device Firmware Upgrade	This example demonstrates Device Firmware Upgrade (DFU) with PSoC™ 6 MCU. For more details, see the README on GitHub .
Emulated EEPROM	This code example demonstrates emulation of EEPROM behavior in flash memory of PSoC™ 6 MCU using emeprom middleware. For more details, see the README on GitHub .
MCUboot-Based Basic Bootloader	This code example demonstrates building a simple bootloader application for PSoC™ 6 MCUs using the open-source library - MCUboot. For more details, see the README on GitHub .
Littlefs Filesystem	This example shows how to perform file I/O operations using the littlefs filesystem API on storage devices such as SD card and NOR flash. This example uses the mtb-littlefs library that implements the block device drivers for use with littlefs. For more details, see the README on GitHub .
emFile Filesystem	This example shows how to create a FAT filesystem on storage devices such as SD card and NOR flash using the SEGGER emFile library. For more details, see the README on GitHub .
QSPI XIP	This example demonstrates how to use the QSPI block of the PSoC™ 6 MCU in execute-in-place (XIP) mode with an external flash memory device. For more details, see the README on GitHub .
QSPI Flash Read Write Using SFDP	This example demonstrates interfacing with an external NOR flash memory in Quad-SPI mode using the Serial Memory Interface (SMIF) block in PSoC™ 6 MCU. This example uses the Serial Flash Discoverable Parameters (SFDP) standard to auto-discover the flash parameters and the commands for read, program, and erase operations. For more details, see the README on GitHub .
Serial Flash Read Write	This example demonstrates interfacing with an external NOR flash memory using the serial flash library. For more details, see the README on GitHub .
USB CDC echo	This example demonstrates how to configure the USB block in a PSoC™ 6 MCU for communication device class (CDC). When configured for a CDC, the PSoC™ 6 MCU device enumerates a virtual COM port, which can be read and written by a terminal emulator program on a PC. For more details, see the README on GitHub .
CSD Current DAC	This code example demonstrates the usage of CAPSENSE™ Sigma Delta (CSD) current digital-to-analog converter (IDAC) in PSoC™ 6 MCU. It loops through the configured current settings and then drives the external pin as per the setting which can be verified through an external load resistor. The configured current setting is also displayed in the terminal application. For more details, see the README on GitHub .
emUSB-Device CDC echo	This example demonstrates how to configure the USB block in a PSoC™ 6 MCU for communication device class (CDC) using Segger's emUSB-Device middleware. When configured for a CDC, the PSoC™ 6 MCU device gets enumerated as a virtual COM port, which can be read and written by a terminal emulator program on a PC. For more details, see the README on GitHub .

4 PSoC™ 6 code examples

DRAFT	This example demonstrates how to configure the USB block in a PSoC™ 6 MCU as a Human Interface Device (HID) using Segger's emUSB-Device middleware. The device enumerates as a Generic HID device. For more details, see the README on GitHub .
emUSB-Device HID Mouse	This example demonstrates how to configure the USB block in a PSoC™ 6 MCU as a HID using Segger's emUSB-Device middleware. The device enumerates as a 3-button mouse. For more details, see the README on GitHub .
emUSB-Device Mass Storage File System	This example demonstrates how to configure the USB block in a PSoC™ 6 MCU device as a Mass Storage (MSC) device and run a file system (FatFs) through an external memory (microSD). This example uses FreeRTOS and Segger's emUSB-Device middleware. For more details, see the README on GitHub .
emUSB-Device Suspend and Resume	This code example demonstrates how PSoC™ 6 USB detects a suspend condition, enters a low-power state, and restores normal operation when USB activity is resumed. The code example uses Segger's emUSB-Device middleware. For more details, see the README on GitHub .
emUSB-Host CDC echo	This example demonstrates how to configure the USB block in a PSoC™ 6 MCU as a communication device class (CDC) host using Segger's emUSB-Host middleware. When configured as a CDC host, the PSoC™ 6 MCU device enumerates the CDC device as a virtual COM port and echoes back the incoming data from the device. For more details, see the README on GitHub .
HAL I2C Slave	This example demonstrates the use of I2C (HAL) resource in Slave mode. For more details, see the README on GitHub .
LZ4 compression and decompression demo	This code example demonstrates a simple LZ4 compression and decompression on a byte array. The byte array included with this application is the pre-built image of the README on GitHub . application. The application boots the decompressed byte array on a user command. For more details, see the README on GitHub .
Audio Streaming	This example, which is based on FreeRTOS, demonstrates how to stream audio data, with a publicly available and widely used protocol Constant Overhead Byte Stuffing (COBS) , from a micro controller to a host system. For more details, see the README on GitHub .
HAL SPI Slave	This example demonstrates the use of SPI (HAL) resource in Slave mode. For more details, see the README on GitHub .
CAN FD	This example demonstrates how to use CAN FD in Infineon's PSoC™ 6 MCU devices. In this example, the CANFD Node-1 sends a CAN FD frame to CANFD-Node-2 on pressing the user button and vice versa. Both the CAN FD nodes log the received data over UART serial terminal. Each time a CAN FD Frame is received, the user LED toggles. For more details, see the README on GitHub .

Sensing

CAPSENSE Buttons and Slider	This code example features a 5-segment CAPSENSE™ slider and two CAPSENSE™ buttons. Button 0 turns the LED on, Button 1 turns the LED off and the slider controls the brightness of the LED. The code example also demonstrates interfacing with Tuner GUI using I2C interface. For more details, see the README on GitHub .
CAPSENSE Buttons and Slider FreeRTOS	This code example features a 5-segment CAPSENSE™ slider and two CAPSENSE™ buttons. Button 0 turns the LED on, Button 1 turns the LED off and the slider controls the brightness of the LED. The code example also demonstrates interfacing with Tuner GUI using I2C interface. For more details, see the README on GitHub .

~~4 PSoC™ 6 code examples~~

SAR ADC Low Power Sensing - Thermistor and ALS	This example demonstrates low-power sensing of a thermistor and ambient light sensor (ALS) using the SAR ADC of PSoC™ 6 MCU. PDL is used for the application firmware. This example is supported only for the devices like CY8C62x4 which have a SAR ADC capable of operating in System Deep Sleep mode. For more details, see the README on GitHub .
CAPSENSE Tuning Over Bluetooth LE - Server	This code example demonstrates how to monitor the CAPSENSE™ data and tune the CAPSENSE™ sensors over Bluetooth® Low Energy communication with CAPSENSE™ Tuner GUI using PSoC™ 6 Bluetooth® Low Energy MCU. This is intended to be used with the CAPSENSE™ Tuning Over Bluetooth® LE - Client example. For more details, see the README on GitHub .
CAPSENSE Tuning Over Bluetooth LE - Client	This code example demonstrates how to monitor the CAPSENSE™ data and tune the CAPSENSE™ sensors over Bluetooth® Low Energy communication with CAPSENSE™ Tuner GUI using PSoC™ 6 Bluetooth® Low Energy MCU. This is intended to be used with the CAPSENSE™ Tuning Over Bluetooth® LE - Server example. For more details, see the README on GitHub .
SAR ADC Simultaneous Sampling	This code example demonstrates simultaneous sampling of two SAR ADCs. The simultaneously sampled input voltages by SAR ADCs are displayed on UART. PDL is used for the application firmware. This example is supported only for the devices like CY8C62x4 which have two SAR ADCs. For more details, see the README on GitHub .
CAPSENSE Custom Scan	This code example demonstrates CAPSENSE™ custom scanning through CAPSENSE™ Middleware's callback functions that allow altering the sensor parameters during runtime or synchronizing the CAPSENSE™ scan with non-CAPSENSE#8482 operations. In this code example, the callback function is used to change the inactive sensor state to either shield or ground depending on the sensor being scanned. For more details, see the README on GitHub .
Low-power CAPSENSE FreeRTOS	This code example demonstrates how to create a low-power CAPSENSE™ design using PSoC™ 6 MCU. This example features a 5-segment CAPSENSE™ slider and a Ganged Sensor, and displays the detected touch position over the serial terminal. For more details, see the README on GitHub .
BMI160 Motion Sensor over I2C FreeRTOS	This code example demonstrates interfacing of the BMI160 Motion Sensor with PSoC™ 6 MCU over an I2C interface within a FreeRTOS task. This example reads the raw motion data and estimates the orientation of the board. For more details, see the README on GitHub .
Human Presence Detection	This application demonstrates the use of the radar presence detection. For more details, see the README on GitHub .
CAPSENSE on CM0p	This code example demonstrates how to create a CAPSENSE#8482; design using CM0+ CPU of PSoC#8482; 6 MCU, without utilizing CM4 CPU. The code example features CAPSENSE#8482; buttons, button 0 turns the LED ON while the button 1 turns the LED OFF and the slider controls the brightness of the LED using the CAPSENSE#8482; Middleware Library. The code example also demonstrates interfacing with Tuner GUI using I2C interface. For more details, see the README on GitHub .

Graphics

emWin E-Ink FreeRTOS	This code example demonstrates displaying 2D graphics on an E-Ink display using EmWin graphics library in FreeRTOS. For more details, see the README on GitHub .
emWin OLED FreeRTOS	This code example demonstrates displaying 2D graphics on an OLED display using EmWin graphics library in FreeRTOS. For more details, see the README on GitHub .

4 PSoC™ 6 code examples

DRAFT	<p>emWin TFT FreeRTOS</p> <p>This example demonstrates displaying 2D graphics on a TFT display using the Segger emWin graphics library and the AppWizard GUI design tool in FreeRTOS. The application initializes the system peripherals and creates a task that cycles through demo images in response to button presses. For more details, see the README on GitHub.</p>
Bluetooth®	
Bluetooth LE Multi Beacon	This code example demonstrates the implementation of a beacon which advertises Eddystone and iBeacon UUID data. For more details, see the README on GitHub .
Wi-Fi Onboarding Using Bluetooth LE	This code example demonstrates Wi-Fi Onboarding using Bluetooth® Low Energy. For more details, see the README on GitHub .
Bluetooth LE Battery Server	This code example demonstrates the implementation of a simple Bluetooth® LE Battery Service. The Battery Service exposes the Battery Level of the device and comes with support for OTA update over Bluetooth® LE. For more details, see the README on GitHub .
Bluetooth LE CAPSENSE Buttons and Slider	This code example features a 5-segment CAPSENSE™ slider and two CAPSENSE™ buttons with Bluetooth® Low Energy custom service. Button 0 turns the LED on, Button 1 turns the LED off and the slider controls the brightness of the LED. The LED status is notified to the client via Bluetooth® Low Energy GATT profile. For more details, see the README on GitHub .
Bluetooth LE Find Me	This code example demonstrates the implementation of a simple Bluetooth® Low Energy Immediate Alert Service (IAS)-based Find Me Profile (FMP) using PSoC™ 6 MCU with Bluetooth® Low Energy Connectivity. For more details, see the README on GitHub .
Bluetooth LE Peripheral Privacy	This code example demonstrates the privacy features available to users in Bluetooth® Low Energy 5.0 and above using ModusToolbox™ software. For more details, see the README on GitHub .
Bluetooth LE IoT Gateway	This code example demonstrates implementing a Bluetooth® IoT gateway using the MQTT client library and Bluetooth® LE host stack for Infineon connectivity devices. The MQTT client library uses the AWS IoT device SDK that includes an MQTT 3.1.1 client as well as libraries specific to AWS IoT such as Thing Shadows. The Bluetooth® Mesh stack runs with FreeRTOS in the server and client model. For more details, see the README on GitHub . License Disclaimer: This code example makes use of the lwIP open-source TCP/IP stack. Creating a project from this template will cause lwIP to be downloaded on your computer. It is your responsibility to understand and accept the lwIP license. Creating a project from this template will cause Mbed TLS to be downloaded on your computer. It is your responsibility to understand and accept the Mbed TLS license and regional use restrictions (including abiding by all applicable export control laws).

Wi-Fi

Wi-Fi Scan	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. For more details, see the README on GitHub . 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 computer. It is your responsibility to understand and accept the lwIP & Mbed TLS licenses and regional use restrictions (including abiding by all applicable export control laws).
----------------------------	---

4 PSoC™ 6 code examples

DRAFT

WPS Enrollee	This code example demonstrates how to use the connection management and WPS Enrollee feature provided in the Wi-Fi Connection Manager (WCM) middleware. For more details, see the README on GitHub . 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 computer. It is your responsibility to understand and accept the lwIP & Mbed TLS licenses and regional use restrictions (including abiding by all applicable export control laws).
WLAN Low Power	This code example demonstrates the low-power operation of a host MCU and a WLAN device using the network activity handlers provided by the Low Power Assistant middleware for AIROC™ Wi-Fi & Bluetooth® combos. For more details, see the README on GitHub . License Disclaimer: This code example makes use of the lwIP open-source TCP/IP stack. Creating a project from this template will cause lwIP to be downloaded on your computer. It is your responsibility to understand and accept the lwIP license. Creating a project from this template will cause Mbed TLS to be downloaded on your computer. It is your responsibility to understand and accept the Mbed TLS license and regional use restrictions (including abiding by all applicable export control laws).
Tester - Wi-Fi Cert Tool	This Tester is a Wi-Fi Cert tool used for Wi-Fi 11n PSK/Enterprise Security certification for PSoC™ 6 MCU with 2 MB flash and CYW43xxx connectivity devices. For more details, see the README on GitHub . License Disclaimer: This tester application makes use of the lwIP open-source TCP/IP stack. Creating a project from this template will cause lwIP to be downloaded on your computer. It is your responsibility to understand and accept the lwIP license. Creating a project from this template will cause Mbed TLS to be downloaded on your computer. It is your responsibility to understand and accept the Mbed TLS license and regional use restrictions (including abiding by all applicable export control laws).
TCP Keepalive Offload	This code example demonstrates the TCP Keepalive Offload functionality offered by AIROC™ Wi-Fi & Bluetooth® combos using PSoC™ 6 MCU. It employs Low Power Assistant (LPA) middleware library which helps in developing low power applications for the Infineon devices. For more details, see the README on GitHub . License Disclaimer: This code example makes use of the lwIP open-source TCP/IP stack. Creating a project from this template will cause lwIP to be downloaded on your computer. It is your responsibility to understand and accept the lwIP license. Creating a project from this template will cause Mbed TLS to be downloaded on your computer. It is your responsibility to understand and accept the Mbed TLS license and regional use restrictions (including abiding by all applicable export control laws).
Wi-Fi UDP Client	This code example demonstrates implementation of UDP Client using the Wi-Fi connectivity SDK. The UDP Client establishes a connection with a remote UDP server and based on the command received from the UDP server, turns the user LED ON or OFF using PSoC™ 6 MCU. For more details, see the README on GitHub . License Disclaimer: This code example makes use of the lwIP open-source TCP/IP stack. Creating a project from this template will cause lwIP to be downloaded on your computer. It is your responsibility to understand and accept the lwIP license. Creating a project from this template will cause Mbed TLS to be downloaded on your computer. It is your responsibility to understand and accept the Mbed TLS license and regional use restrictions (including abiding by all applicable export control laws).

4 PSoC™ 6 code examples

DRAFT

Wi-Fi UDP Server	This code example demonstrates implementation of UDP Server using the Wi-Fi connectivity SDK. The UDP Server allows the user to send LED ON/OFF command to the UDP client using PSoC™ 6 MCU. For more details, see the README on GitHub . License Disclaimer: This code example makes use of the lwIP open-source TCP/IP stack. Creating a project from this template will cause lwIP to be downloaded on your computer. It is your responsibility to understand and accept the lwIP license. Creating a project from this template will cause Mbed TLS to be downloaded on your computer. It is your responsibility to understand and accept the Mbed TLS license and regional use restrictions (including abiding by all applicable export control laws).
OTA Using HTTPS	This code example demonstrates OTA update with PSoC™ 6 MCU and AIROC™ CYW43xxx Wi-Fi & Bluetooth® combo chips. The device establishes a connection with the designated HTTPS server. It periodically checks the job document to see if a new update is available. When a new update is available, it is downloaded and written to the secondary slot. On the next reboot, MCUBoot swaps the new image in the secondary slot with the primary slot image and runs the application. If the new image is not validated in runtime, on the next reboot, MCUBoot reverts to the previously validated image. For more details, see the README on GitHub . License Disclaimer: This code example makes use of the lwIP open-source TCP/IP stack. Creating a project from this template will cause lwIP to be downloaded on your computer. It is your responsibility to understand and accept the lwIP license. Creating a project from this template will cause Mbed TLS to be downloaded on your computer. It is your responsibility to understand and accept the Mbed TLS license and regional use restrictions (including abiding by all applicable export control laws).
OTA Using MQTT	This code example demonstrates OTA update with PSoC™ 6 MCU and AIROC™ CYW43xxx Wi-Fi & Bluetooth® combo chips. The device establishes a connection with the designated MQTT broker (this example uses a local Mosquitto broker). It periodically checks the job document to see if a new update is available. When a new update is available, it is downloaded and written to the secondary slot. On the next reboot, MCUBoot swaps the new image in the secondary slot with the primary slot image and runs the application. If the new image is not validated in runtime, on the next reboot MCUBoot reverts to the previously validated image. For more details, see the README on GitHub . License Disclaimer: This code example makes use of the lwIP open-source TCP/IP stack. Creating a project from this template will cause lwIP to be downloaded on your computer. It is your responsibility to understand and accept the lwIP license. Creating a project from this template will cause Mbed TLS to be downloaded on your computer. It is your responsibility to understand and accept the Mbed TLS license and regional use restrictions (including abiding by all applicable export control laws).
Wi-Fi HTTPS Server	This code example demonstrates the implementation of an HTTPS server with PSoC™ 6 MCU and CYW43xxx connectivity devices. It employs the HTTPS server middleware library, which takes care of all the underlying socket connections with the HTTPS client. This example establishes a secure connection with a HTTPS client through SSL handshake. Once the SSL handshake completes successfully, the HTTPS client can make GET, POST, and PUT requests with the server. For more details, see the README on GitHub . License Disclaimer: This code example makes use of the lwIP open-source TCP/IP stack. Creating a project from this template will cause lwIP to be downloaded on your computer. It is your responsibility to understand and accept the lwIP license. Creating a project from this template will cause Mbed TLS to be downloaded on your computer. It is your responsibility to understand and accept the Mbed TLS license and regional use restrictions (including abiding by all applicable export control laws).

4 PSoC™ 6 code examples

DRAFT

Wi-Fi Web Server	This code example demonstrates Wi-Fi provisioning via a SoftAP interface and setting up a web server using PSoC™ 6 MCU with AIROC™ CYW43xxx Wi-Fi & Bluetooth® combo chips. In this example, the PSoC™ 6 MCU device is configured in AP+STA concurrent mode. It starts an HTTP web server while in AP+STA concurrent mode and hosts an HTTP web page. The kit can be connected to the desired Wi-Fi network by entering the credentials via this web page. After connecting to the specified Wi-Fi network, the device is reconfigured to start a new HTTP web server. The new web server hosts a web page that displays the device data containing ambient light sensor (ALS) value and provides a button to change the brightness of an LED. For more details, see the README on GitHub . 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 computer. It is your responsibility to understand and accept the lwIP & Mbed TLS licenses and regional use restrictions (including abiding by all applicable export control laws).
Wi-Fi Secure TCP client	This code example demonstrates the implementation of a secure TCP client using PSoC™ 6 MCU with AIROC™ CYW43xxx Wi-Fi & Bluetooth® combo chips. For more details, see the README on GitHub . 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 computer. It is your responsibility to understand and accept the lwIP & Mbed TLS licenses and regional use restrictions (including abiding by all applicable export control laws).
Wi-Fi Secure TCP server	This code example demonstrates the implementation of a secure TCP server using PSoC™ 6 MCU with AIROC™ CYW43xxx Wi-Fi & Bluetooth® combo chips. For more details, see the README on GitHub . 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 computer. It is your responsibility to understand and accept the lwIP & Mbed TLS licenses and regional use restrictions (including abiding by all applicable export control laws).
Wi-Fi TCP Client	This code example demonstrates implementation of TCP client with PSoC™ 6 MCU and CYW43xxx connectivity devices. For more details, see the README on GitHub . License Disclaimer: This code example makes use of the lwIP open-source TCP/IP stack. Creating a project from this template will cause lwIP to be downloaded on your computer. It is your responsibility to understand and accept the lwIP license. Creating a project from this template will cause Mbed TLS to be downloaded on your computer. It is your responsibility to understand and accept the Mbed TLS license and regional use restrictions (including abiding by all applicable export control laws).
Wi-Fi TCP Server	This code example demonstrates implementation of a TCP server with PSoC™ 6 MCU and CYW43xxx connectivity devices. For more details, see the README on GitHub . 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 computer. It is your responsibility to understand and accept the lwIP & Mbed TLS licenses and regional use restrictions (including abiding by all applicable export control laws).

4 PSoC™ 6 code examples

DRAFT

Wi-Fi MQTT Client	This code example demonstrates implementing an MQTT Client using the MQTT Client library. The library uses the AWS IoT device SDK Port library and implements the glue layer that is required for the library to work with Infineon™ connectivity platforms. This example can be ported to CM0+ core using a make variable CORE from Makefile. For more details, see the README on GitHub . License Disclaimer: This code example makes use of the lwIP open-source TCP/IP stack. Creating a project from this template will cause lwIP to be downloaded on your computer. It is your responsibility to understand and accept the lwIP license. Creating a project from this template will cause Mbed TLS to be downloaded on your computer. It is your responsibility to understand and accept the Mbed TLS license and regional use restrictions (including abiding by all applicable export control laws).
Connecting to Azure IoT	This code example demonstrates connecting to the Azure IoT services using the Azure SDK for Embedded C and Infineon's Wi-Fi connectivity SDK in ModusToolbox™. This code example demonstrates the features such as IoT hub - C2D (Cloud-to-Device messaging), Telemetry, Methods, Twin, Provisioning, and PnP (Plug and Play). For more details, see the README on GitHub . License Disclaimer: This code example makes use of the lwIP open-source TCP/IP stack. Creating a project from this template will cause lwIP to be downloaded on your computer. It is your responsibility to understand and accept the lwIP license. Creating a project from this template will cause Mbed TLS to be downloaded on your computer. It is your responsibility to understand and accept the Mbed TLS license and regional use restrictions (including abiding by all applicable export control laws).
Wi-Fi MCUboot-Based Bootloader with Rollback	This code example implements a bootloader based on MCUboot to demonstrate ‘Rollback’ to a known good image ('factory_app_cm4') in case of unrecoverable error conditions in the current application. The bootloader can load the factory app from a known location in the external memory by directly copying it into the primary slot in the internal flash, based on user inputs during boot. The factory app can then perform the OTA to download an image over Wi-Fi and place it to the secondary slot of MCUboot. For more details, see the README on GitHub . License Disclaimer: This code example makes use of the lwIP open-source TCP/IP stack. Creating a project from this template will cause lwIP to be downloaded on your computer. It is your responsibility to understand and accept the lwIP license. Creating a project from this template will cause Mbed TLS to be downloaded on your computer. It is your responsibility to understand and accept the Mbed TLS license and regional use restrictions (including abiding by all applicable export control laws).
AWS IoT OTA Using MQTT	This code example demonstrates an OTA update with PSoC™ 6 MCU and AIROC™ CYW43xxx Wi-Fi & Bluetooth combo chips. The example uses the "AWS SDK for Embedded C" and "AWS IoT device sdk port" to connect and communicate with the AWS IoT MQTT core. For more details, see the README on GitHub . License Disclaimer: This code example makes use of the lwIP open-source TCP/IP stack. Creating a project from this template will cause lwIP to be downloaded on your computer. It is your responsibility to understand and accept the lwIP license. Creating a project from this template will cause Mbed TLS to be downloaded on your computer. It is your responsibility to understand and accept the Mbed TLS license and regional use restrictions (including abiding by all applicable export control laws).
CCM hello world	This code example demonstrates publishing MQTT messages to the AWS IoT Core with the help of the Cloud Connectivity Manager (CCM) evaluation kit. For more details, see the README on GitHub .

4 PSoC™ 6 code examples

CCM CAPSENSE Publish	This code example demonstrates publishing MQTT messages to AWS IoT Core with the help of Cloud Connectivity Manager (CCM) evaluation kit. This code example features a 5-segment CAPSENSE™ slider. The brightness of the LED is calculated based on the position in linear slider and the brightness is published to the topic "MySubTopic" in AWS IoT core. For more details, see the README on GitHub .
CCM Subscribe OTA	This code example demonstrates MQTT Subscribe and OTA from AWS IoT Core with the help of the Cloud Connectivity Manager (CCM) evaluation kit. For more details, see the README on GitHub .

Manufacturing

Tester - Bluetooth MFG Tool	This application is used to validate the Bluetooth® Firmware and RF performance for PSoC™ 6 MCU with Bluetooth® BR/EDR/LE devices. For more details, see the README on GitHub .
Tester - Wi-Fi MFG Tool	This tester is a tool used for Wi-Fi manufacturing tests with PSoC™ 6 MCU and CYW43xxx connectivity devices. For more details, see the README on GitHub . License Disclaimer: This tester application makes use of the lwIP open-source TCP/IP stack. Creating a project from this template will cause lwIP to be downloaded on your computer. It is your responsibility to understand and accept the lwIP license. Creating a project from this template will cause Mbed TLS to be downloaded on your computer. It is your responsibility to understand and accept the Mbed TLS license and regional use restrictions (including abiding by all applicable export control laws).

Connectivity

Matter Wi-Fi Door Lock	This code example makes use of Matter on an Infineon PSoC™ 6 board to demonstrate its functionality as a Wi-Fi door lock device. For more details, see the README on GitHub .
------------------------	---

Machine Learning

Machine Learning Neural Network Profiler	This code example demonstrates how to run through the machine learning (ML) development flow with PSoC™ 6 MCU, where the end user has a pre-trained Neural Network (NN) model, which can be profiled and validated at the PC and target device. For more details, see the README on GitHub .
Machine Learning Gesture Classification	This code example demonstrates how to perform gesture classification based on motion sensor (accelerometer and gyroscope) data. The code example comes with a pre-trained model that classifies the following gestures: circle, square, and side-to-side. For more details, see the README on GitHub .
SensiML Template Firmware	This package contains a demo project for the CY8CKIT-062S2-43012 and CY8CKIT-028-SENSE kit using SensiML. This application is to collect data allowing models to be generated using the SensiML platform and deployed to the device. For more details, see the README on GitHub . SensiML, a subsidiary of QuickLogic, offers cutting-edge AutoML software tools empowering application developers to rapidly build smart IoT devices, transforming raw sensor data into autonomous meaningful insight. > Get Started > SensiML Subscription Plans

5 PSoC™ 6 application notes

5 PSoC™ 6 application notes

Application notes cover a broad range of topics, from basic to advanced level concepts in the PSoC™ 6 platform. This section includes all the application notes available for PSoC™ 6 platform.

5.1 AN228571 Getting started with PSoC™ 6 MCU on ModusToolbox™ software

About this document

- 1

Scope and purpose

This application note introduces the PSoC™ 6 MCU, a dual-core programmable system-on-chip with Arm® Cortex®-M4 and Cortex®-M0+ processors. This application note helps you explore the PSoC™ 6 MCU architecture and development tools and shows you how to create your first application using ModusToolbox™ software. This application note also guides you to more resources available online to accelerate your learning about PSoC™ 6 MCU.

Intended audience

This document is intended for the users who are new to PSoC™ 6 MCU and ModusToolbox™ software.

Associated part family

All PSoC™ 6 MCU devices

Software version

ModusToolbox™ software 3.0 or above

More code examples? We heard you.

To access an ever-growing list of PSoC™ code examples using ModusToolbox™, please visit the [GitHub](#) site. You can also explore the [PSoC™ video library](#).

~~DEAF~~ 5 PSoC™ 6 application notes

5.1.1 Introduction

PSoC™ 6 MCU is an ultra-low-power PSoC™ device with a dual-core architecture tailored for smart homes, IoT gateways, etc. The PSoC™ 6 MCU is a programmable embedded system-on-chip that integrates the following features on a single chip:

- Single-core microcontroller: Arm® Cortex®-M4 (CM4); or dual-core microcontroller: Arm® Cortex®-M4 (CM4) and Cortex®-M0+ (CM0+)
- Programmable analog and digital peripherals
- Up to 2 MB of flash and 1 MB of SRAM
- Fourth-generation CAPSENSE™ technology
- PSoC™ 6 MCU is suitable for a variety of power-sensitive applications such as:
 - Smart home sensors and controllers
 - Smart home appliances
 - Gaming controllers
 - Sports, smart phone, and virtual reality (VR) accessories
 - Industrial sensor nodes
 - Industrial logic controllers
 - Advanced remote controllers
 - Wearables

The [ModusToolbox™ software environment](#) supports PSoC™ 6 MCU application development with a set of tools for configuring the device, setting up peripherals, and complementing your projects with world-class middleware. See the [Infineon GitHub repos](#) for BSPs (Board Support Packages) for all kits, libraries for popular functionality like CAPSENSE™ and emWin, and a comprehensive array of [example applications](#) to get you started.

Figure 3 illustrates an application-level block diagram for a real-world use case using PSoC™ 6 MCU.

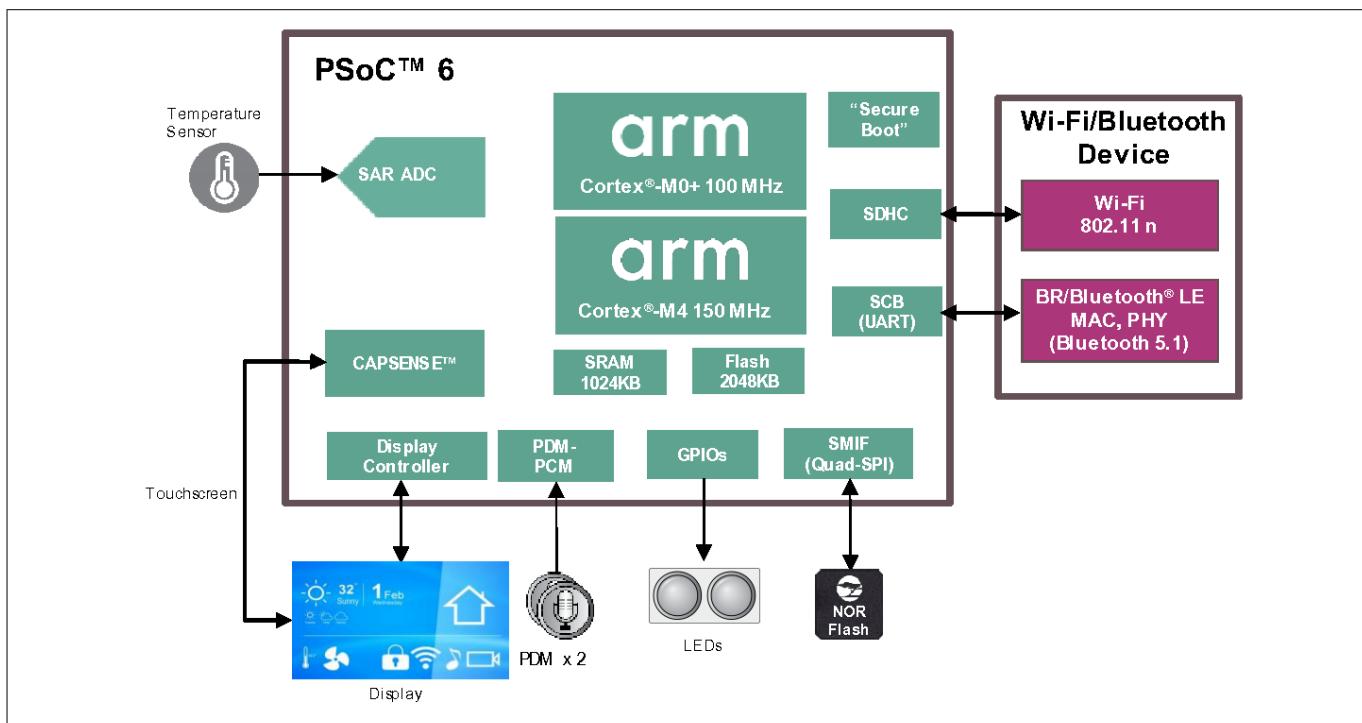


Figure 3 Application-level block diagram using PSoC™ 6 MCU

PSoC™ 6 MCU is a highly capable and flexible solution. For example, the real-world use case in [Figure 3](#) takes advantage of these features:

~~5 PSoC™ 6 application notes~~

- A buck converter for ultra-low-power operation
- An analog front end (AFE) within the device to condition and measure sensor outputs such as ambient light sensor
- Serial Communication Blocks (SCBs) to interface with multiple digital sensors such as motion sensors
- CAPSENSE™ technology for reliable touch and proximity sensing
- Programmable Digital logic (Smart I/O) and peripherals (Timer Counter PWM or TCPWM) to drive the display and LEDs respectively
- SDIO interface to a Wi-Fi/Bluetooth® device to provide IoT cloud connectivity
- Product security features managed by CM0+ core and application features executed by CM4 core

There are four product lines in PSoC™ 6 which cater to different application needs. [Table 1](#) provides overview of different product lines:

Table 1 PSoC™ 6 MCU product lines

Product line	Security firmware	Device series	Details	Applications
Programmable	No	CY8C61x	Single core: 150-MHz Arm® Cortex®-M4	
Performance	No	CY8C62x	Dual-core architecture: 150-MHz Arm® Cortex®-M4 and 100-MHz Cortex®-M0+	IoT gateways, smart home, home appliances, HMI, audio processing, and industrial concentrators
Connectivity	No	CY8C63x	Dual-core architecture: 150-MHz Arm® Cortex®-M4 and 100-MHz Cortex®-M0+ Bluetooth® low energy (LE) 5.0 radio with 2-Mbps data throughput	Wearables, portable medical, industrial IoT, and smart home
Security	“Secure Boot”	CYB064x	150-MHz Arm® Cortex®-M4 for the user application Hardware isolated, 100-MHz Arm® Cortex®-M0+ with privileged access to memory and peripherals for security functions Bluetooth® LE 5.0 radio with 2-Mbps data throughput Arm® Platform Security Architecture Certifications- PSA L1, FIPS 140-2	
	AWS Standard “Secure”	CYS064x	150-MHz Arm® Cortex®-M4 for the user application Hardware isolated, 100-MHz Arm® Cortex®-M0+ with privileged access to memory and peripherals for security functions Arm® Platform Security Architecture Certifications- PSA L2	IoT gateways, smart home, home appliances, HMI, audio processing, and industrial concentrators

Note that not all the features available in all the devices in a product line. See the [device datasheets](#) for more details.

This application note introduces you to the capabilities of the PSoC™ 6 MCU, gives an overview of the development ecosystem, and gets you started with a simple ‘Hello World’ application wherein you learn to use the PSoC™ 6 MCU. We will show you how to create the application from an empty starter application, but the completed design is available as a [code example for ModusToolbox™ on GitHub](#).

5 PSoC™ 6 application notes

For hardware design considerations, see [AN218241 – PSoC™ 6 MCU hardware design considerations](#).

DRAFT

5 PSoC™ 6 application notes

5.1.2 Development ecosystem

5.1.2.1 PSoC™ resources

A wealth of data available [here](#) helps you to select the right PSoC™ MCU and quickly and effectively integrate it into your design. For a comprehensive list of PSoC™ 6 MCU resources, see [How to design with PSoC™ 6 MCU - KBA223067](#). The following is an abbreviated list of resources for PSoC™ 6 MCU.

- Overview: [PSoC™ portfolio](#)
- [PSoC™ 6 MCU webpage](#)
- Product selectors: [PSoC™ 6 MCU](#)
- [Datasheets](#) describe and provide electrical specifications for each device family.
- [Application notes](#) and [Code examples](#) cover a broad range of topics, from basic to advanced level. You can also browse our collection of code examples.
- [Technical reference manuals \(TRMs\)](#) provide detailed descriptions of the architecture and registers in each device family.
- [PSoC™ 6 MCU programming specification](#) provides the information necessary to program the nonvolatile memory of PSoC™ 6 MCU devices.
- [CAPSENSE™ design guides](#): Learn how to design capacitive touch-sensing applications with PSoC™ devices.
- Development tools: Many low-cost [kits and shield boards](#) are available for evaluation, design, and development of different applications using PSoC™ 6 MCUs.
- Training videos: [Video training](#) on our products and tools, including a dedicated series on [PSoC™ 6 MCUs](#).
- Technical Support: [PSoC™ 6 community forum](#), [Knowledge base articles](#).

5.1.2.2 Firmware/application development

There are two development platforms that you can use for application development with PSoC™ 6 MCU:

- **ModusToolbox™**: This software includes configuration tools, low-level drivers, middleware libraries and other packages that enable you to create MCU and wireless applications. All tools run on Windows, macOS and Linux. ModusToolbox™ includes an Eclipse IDE, which provides an integrated flow with all the ModusToolbox™ tools. Other IDEs such as Visual Studio Code, IAR Embedded Workbench and Arm® MDK (μ Vision®) are also supported.

ModusToolbox™ software supports stand-alone device and middleware configurators. Use the configurators to set the configuration of different blocks in the device and generate code that can be used in firmware development. The software supports all PSoC™ 6 MCUs. It is recommended that you use ModusToolbox™ software for all application development for PSoC™ 6 MCUs. See the [ModusToolbox™ tools package user guide](#) for more information.

Libraries and enablement software are available at the [GitHub](#) site.

Software resources available at GitHub support one or more of the target ecosystems:

- MCU and Bluetooth® SoC ecosystem – a full-featured platform for PSoC™ 6 MCU, Bluetooth®, and Bluetooth® low energy application development.
- Connectivity ecosystem – a set of libraries providing core functionality of Wi-Fi including connectivity, security, firmware upgrade support, and application layer protocols for applications.
- Amazon FreeRTOS ecosystem – extends the FreeRTOS kernel with software libraries that make it easy to securely connect small, low-power Infineon devices to most cloud services.

~~5 PSoC™ 6 application notes~~

ModusToolbox™ tools and resources can also be used on the command line. See the build system chapter in the [ModusToolbox™ tools package user guide](#) for detailed documentation.

- **PSoC™ Creator:** A proprietary IDE that runs on Windows only. It supports a subset of PSoC™ 6 MCUs, as well as other PSoC™ families such as PSoC™ 3, PSoC™ 4, and PSoC™ 5LP. See [AN221774 - Getting started with PSoC™ 6 on PSoC™ Creator](#) for more information.

5.1.2.2.1 Installing the ModusToolbox™ tools package

Refer to the [ModusToolbox™ tools package installation guide](#) for details.

5.1.2.2.2 Choosing an IDE

ModusToolbox™ software, the latest-generation toolset, is supported across Windows, Linux, and macOS platforms. ModusToolbox™ software supports 3rd-party IDEs, including the Eclipse IDE, Visual Studio Code, Arm® MDK (μVision), and IAR Embedded Workbench. The tools package includes an implementation for the Eclipse IDE for your convenience. The tools support all PSoC™ 6 MCUs. The associated BSP and library configurators also work on all three host operating systems.

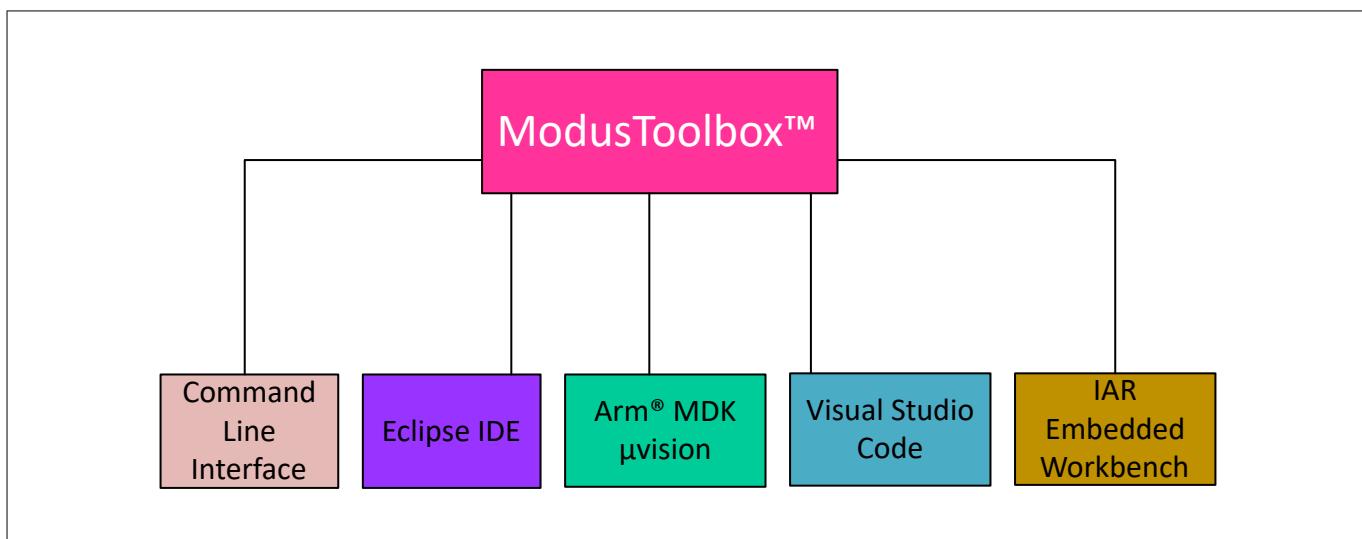


Figure 4 ModusToolbox™ environment

Certain features of the PSoC™ 6 MCU, such as UDBs (Universal Digital Blocks) and USB are not supported in ModusToolbox™ version 2.x and earlier. Newer versions of ModusToolbox™ support the USB host feature and improve the user experience with true multi-core debug support.

It is recommended to use ModusToolbox™ if you want to build an IoT application using IoT devices, or if you are using a PSoC™ 6 MCU not supported in PSoC™ Creator.

PSoC™ Creator is the long-standing proprietary tool that runs on Windows only. This mature IDE includes a graphical editor that supports schematic based design entry with the help of Components. PSoC™ Creator supports all PSoC™ 3, PSoC™ 4, and PSoC™ 5LP devices, and a subset of PSoC™ 6 MCU devices.

Choose PSoC™ Creator if you are inclined towards using a graphical editor for design entry and code generation, and if the PSoC™ MCU that you are planning to use is supported by the IDE or if you are intending to use the UDBs on the PSoC™ MCU.

5.1.2.2.3 ModusToolbox™ software

ModusToolbox™ software is a set of tools and software that enables an immersive development experience for creating converged MCU and wireless systems and enables you to integrate our devices into your existing

5 PSoC™ 6 application notes

~~DRAFT~~
development methodology. These include configuration tools, low-level drivers, libraries, and operating system support, most of which are compatible with Linux-, macOS-, and Windows-hosted environments.

Figure 5 shows a high-level view of what is available as part of ModusToolbox™ software. For a more in-depth overview of the ModusToolbox™ software, see [ModusToolbox™ tools package user guide](#).

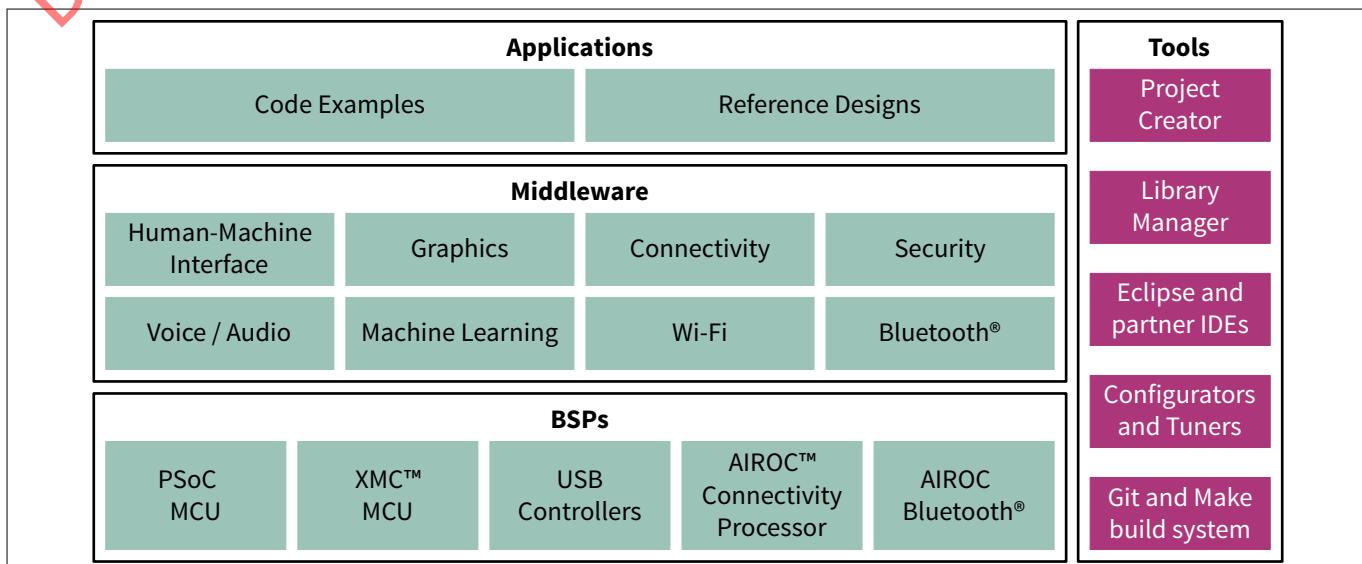


Figure 5 ModusToolbox™ software

The ModusToolbox™ tools package installer includes the design configurators and tools, and the build system infrastructure.

The build system infrastructure includes the new project creation wizard that can be run independent of the Eclipse IDE, the make infrastructure, and other tools. This means you choose your compiler, IDE, RTOS, and ecosystem without compromising usability or access to our industry-leading CAPSENSE™ (Human-Machine Interface), AIROC™ Wi-Fi and Bluetooth®, security, and various other features.

One part of the ModusToolbox™ ecosystem is run-time software that helps you rapidly develop Wi-Fi and Bluetooth® applications using connectivity combo devices, such as AIROC™ CYW43012 and CYW43439 (among others), with the PSoC™ 6 MCU. See the [ModusToolbox™ run-time software reference guide](#) for details.

Design configurators are the tools that help you create the configurable code for your BSP/Middleware. Jump to [Configurators](#) to know more about it.

Figure 6 shows a run-time software diagram to showcase some of the application capabilities of Infineon devices using ModusToolbox™ software.

5 PSoC™ 6 application notes

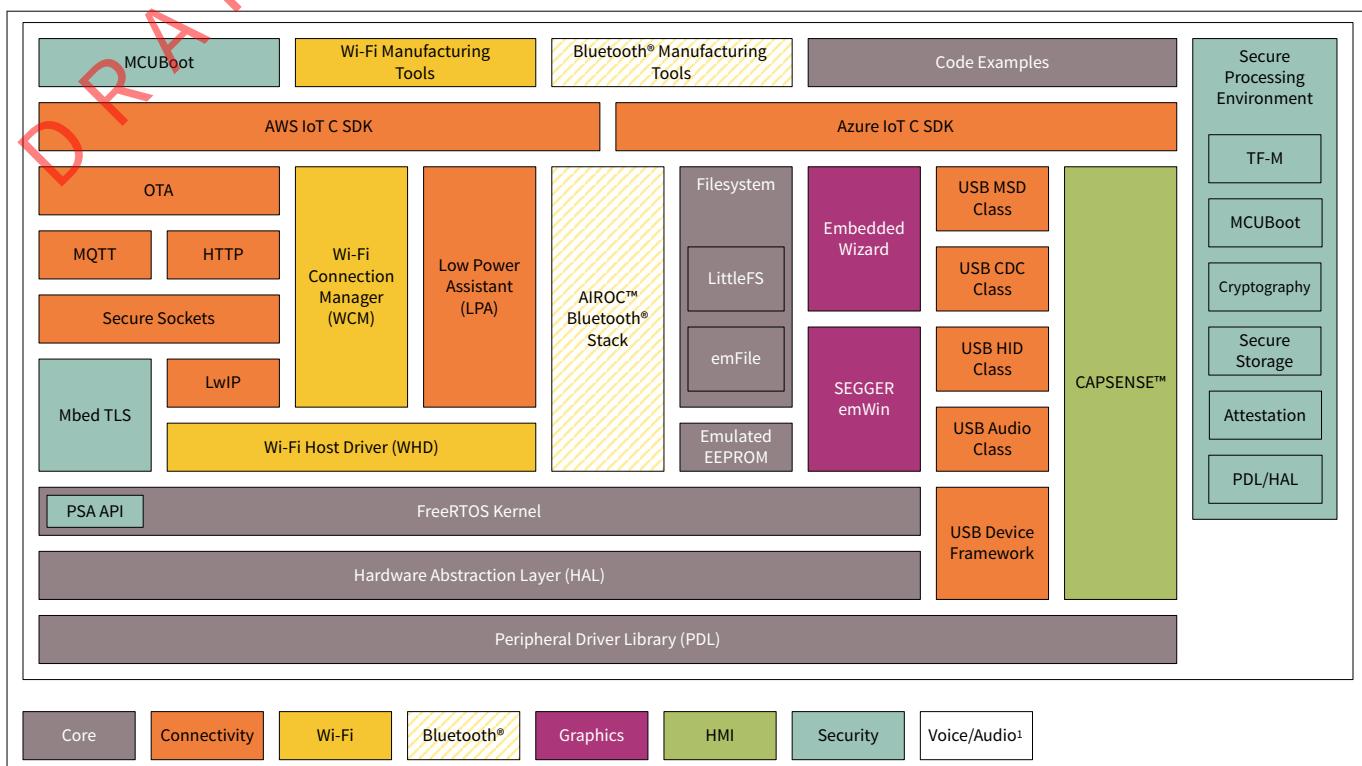


Figure 6 ModusToolbox™ run-time software diagram

All the application-level development flows depend on the provided low-level resources. These include:

- Board support packages (BSP) – A BSP is the layer of firmware containing board-specific drivers and other functions. The BSP is a set of libraries that provide APIs to initialize the board and provide access to board level peripherals. It includes low-level resources such as peripheral driver library (PDL) for PSoC™ 6 MCU and has macros for board peripherals. It uses the HAL to configure the board. Custom BSPs can be created to enable support for end-application boards. See [BSP Assistant](#) to create your BSP.
- [Hardware abstraction layer \(HAL\)](#) – The hardware abstraction layer (HAL) provides a high-level interface to configure and use hardware blocks on 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 wraps the lower level drivers (like PSoC™ 6 PDL) and provides a high-level interface to the MCU. The interface is abstracted to work on any MCU. This helps you write application firmware independent of the target MCU.

The HAL can be combined with platform-specific libraries (such as PSoC™ 6 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 lower-level control.

- [PSoC™ 6 peripheral driver library \(PDL\)](#) – The PDL integrates device header files, start-up code, and peripheral drivers into a single package. The PDL supports the PSoC™ 6 MCU 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 MCU series. You configure the driver for your application, and then use API calls to initialize and use the peripheral.

- Middleware (MW) – Extensive middleware libraries that provides specific capabilities to an application. The [available middleware](#) spans across connectivity (OTA, Bluetooth®, AWS IoT, Bluetooth® LE, Secure Sockets) to PSoC™ 6 MCU-specific functionality (CAPSENSE™, USB, device firmware upgrade (DFU), emWin). All the middleware is delivered as libraries via GitHub repositories.

5 PSoC™ 6 application notes

5.1.2.2.4 ModusToolbox™ applications

With the release of ModusToolbox™ v3.x, multi-core support is introduced, which has altered the folder structure slightly from the previous version of ModusToolbox™.

ModusToolbox™ has two types of applications:

- Single-core application
- Multi-core application

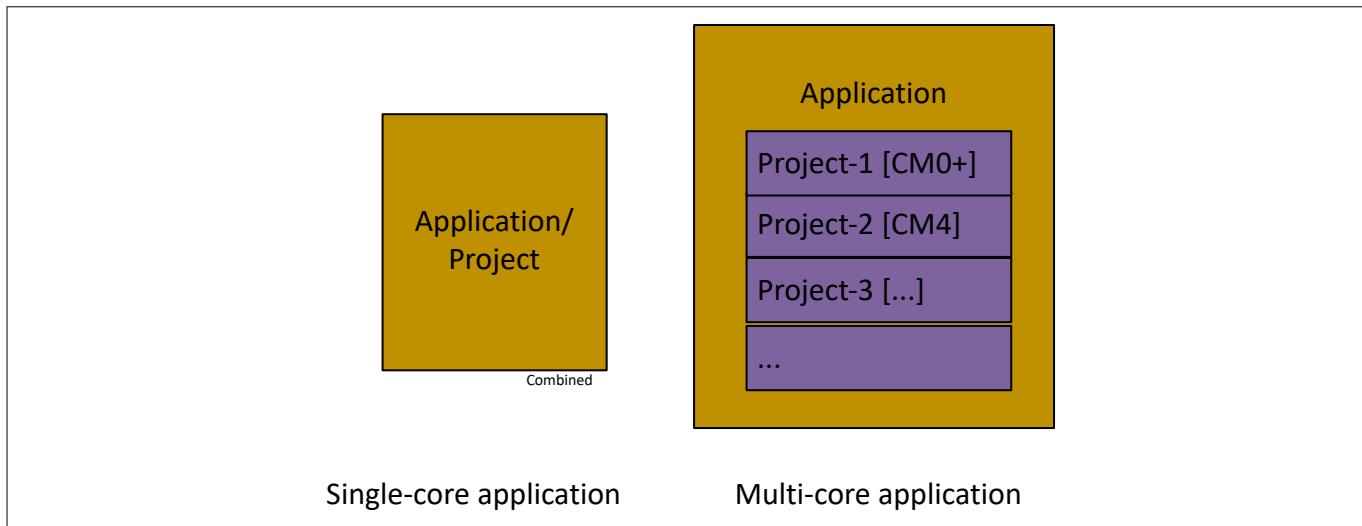


Figure 7

Application types

5 PSoC™ 6 application notes

The following shows the new folder structure for an example single-core application:

```
<root>
  ApplicationName
    ->Makefile (MTB_TYPE=COMBINED)
    ->deps
      lib1.mtb (local)
      lib2.mtb (shared)
    ->libs
      lib1 (Infineon Git repo)
    ->bssps
      TARGET_BSP1 (not an Infineon Git repo; completely app-owned)
    ->templates
      TARGET_BSP1
        design.modus
        design.capsense
    ->main.c
    ->helper.h
    ->helper.c
    mtb_shared
      lib2/... (Infineon Git repo)
```

Figure 8 Folder structure for single-core applications

~~5 PSoC™ 6 application notes~~

The following shows the new folder structure for an example multi-core application:

```

<root>
  ApplicationName
    ->Makefile (MTB_TYPE=APPLICATION)
    ->common.mk
    ->common_app.mk
    ->bssps
      TARGET_BSP1 (not an Infineon Git repo; completely app-owned)
    ->templates
      TARGET_BSP1
        design.modus
        design.capsense
    ->project1
      Makefile (MTB_TYPE=PROJECT)
      deps
        lib3.mtb (local)
        lib4.mtb (shared)
      libs
        lib3 (Infineon Git repo)
      main.c
      project1_helper.h
      project1_helper.c
    ->project2
      Makefile (MTB_TYPE=PROJECT)
      deps
        lib5.mtb (local)
        lib6.mtb (shared)
      libs
        lib5 (Infineon Git repo)
      main.c
      project2_helper.h
      project2_helper.c
    mtb_shared
      lib4/... (Infineon Git repo)
      lib6/... (Infineon Git repo)

```

Figure 9 Folder Structure for multi-core applications

The new flow using ModusToolbox™ versions 3.x can support multiple projects in an application. For multi-core applications, there are multiple projects, but only one project per core. The applications have app-owned BSPs, meaning the BSP will be common to all projects inside a multi-core application.

Going further, section 4 of this document describes creating a new single-core application using ModusToolbox™ software.

5 PSoC™ 6 application notes

5.1.2.2.5 PSoC™ 6 software resources

The software for PSoC™ 6 MCUs includes configurators, drivers, libraries, middleware, as well as various utilities, makefiles, and scripts. It also includes relevant drivers, middleware, and examples for use with IoT devices and connectivity solutions. You may use any or all tools in any environment you prefer.

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 and set the baud rate, parity, and stop bits. Upon saving the hardware configuration, the tool generates the "C" code to initialize the hardware with the desired configuration.

There are two types of configurators: BSP configurators that configure items that are specific to the MCU hardware and library configurators that configure options for middleware libraries.

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 tools, or within a complete IDE. Configurators are used for:

- Setting options and generating code to configure drivers
- Setting up connections such as pins and clocks for a peripheral
- Setting options and generating code to configure middleware

For PSoC™ 6 MCU applications, the available configurators include:

- **Device configurator:** Set up the system (platform) functions, pins, and the basic peripherals (e.g., UART, Timer, PWM).
- **CAPSENSE™ configurator and tuner:** Configure CAPSENSE™ and generate the required code and tune CAPSENSE™ applications.
- **LIN configurator:** Configure LIN middleware and generate the required configuration.
- **ML configurator:** To fit the pre-trained model of choice to the target device with a set of optimization parameters (Only available as a part of separate pack)
- **USB configurator:** Configure USB settings and generate the required code.
- **QSPI configurator:** Configure external memory and generate the required code.
- **Smart I/O configurator:** Configure Smart I/O pins.
- **Bluetooth® configurator:** Configure the Bluetooth® settings.
- **EZ-PD™ configurator:** Configure parameters and select the features of the PD Stack middleware.
- **Secure policy configurator:** Open, create or change policy configuration files for the “Secure” MCU devices.
- **SegLCD configurator:** Configure and generate the required structures for SegLCD driver.

Each of the above configurators create their own files (e.g.: *design.cycapsense* for CAPSENSE™). BSP configurator files (e.g. *design.modus* or *design.cycapsense*) are provided as part of the BSP with default configurations while library configurators (e.g. *design.cybt*) are provided by the application. When an application is created based on Infineon BSP, the application makes use of BSP configurator files from the Infineon BSP repo. You can customize/create all the configurator files as per your application requirement using ModusToolbox™ software. See [BSP Assistant](#) to create your custom BSP. See [ModusToolbox™ help](#) for more details..

5 PSoC™ 6 application notes

~~Library management for PSoC™ 6 MCU~~

The application can have shared/local libraries for the projects. If needed, different projects can use different versions of the same library. The shared libraries are downloaded under the `mtb_shared` directory. The application should use the `deps` folder to add library dependencies. The `deps` folder contains files with the `.mtb` file extension, which is used by ModusToolbox™ to download its git repository. These libraries are direct dependencies of the ModusToolbox™ project.

The Library Manager helps to add/remove/update the libraries of your projects. It also identifies whether the particular library has a direct dependency on any other library using the manifest repository available on GitHub and fetches all the dependencies of that particular library. These dependency libraries are indirect dependencies of the ModusToolbox™ project. These dependencies can be seen under the `libs` folder. For more information, see the [Library Manager user guide](#) located at `<install_dir>/ModusToolbox/tools_<version>/library-manager/docs/library-manager.pdf`.

Software development for PSoC™ 6 MCU

The ModusToolbox™ ecosystem provides significant source code and tools to enable software development for PSoC™ 6 MCUs. You use tools to:

- Specify how you want to configure the hardware
- Generate code for that purpose, which you use in your firmware
- Include various middleware libraries for additional functionality, like Bluetooth® LE connectivity or FreeRTOS

This source code makes it easier to develop the firmware for supported devices. It helps you quickly customize and build firmware without the need to understand the register set.

In the ModusToolbox™ environment, you use configurators to configure either the device, or a middleware library, like the Bluetooth® LE stack or CAPSENSE™. The BSP configurator files are used to configure device peripherals, pins, and memory using peripheral driver library code. The middleware is delivered as separate libraries for each feature/function such that it can be used across multiple platforms. For example, abstraction-rtos, lwip, usb, etc.

Firmware developers who wish to work at the register level should refer to the driver source code from the PDL. The PDL includes all the device-specific header files and startup code you need for your project. It also serves as a reference for each driver. Because the PDL is provided as source code, you can see how it accesses the hardware at the register level.

Some devices do not support particular peripherals. The PDL is a superset of all the drivers for any supported device. This superset design means:

- All API elements needed to initialize, configure, and use a peripheral are available.
- The PDL is useful across various PSoC™ 6 MCUs, regardless of available peripherals.
- The PDL includes error checking to ensure that the targeted peripheral is present on the selected device.

This enables the code to maintain compatibility across members of the PSoC™ 6 MCU family, as long as the peripherals are available. A device header file specifies the peripherals that are available for a device. If you write code that attempts to use an unsupported peripheral, you will get an error at compile time. Before writing code to use a peripheral, consult the datasheet for the particular device to confirm support for that peripheral.

As [Figure 10](#) shows, with the ModusToolbox™ software, you can:

1. Choose a BSP (Project Creator).
2. Create a new application based on a list of starter applications, filtered by the BSPs that each application supports (Project Creator).
3. Add BSP or middleware libraries (Library Manager).
4. Develop your application firmware using the HAL or PDL for PSoC™ 6 MCU (IDE of choice or command line).

5 PSoC™ 6 application notes

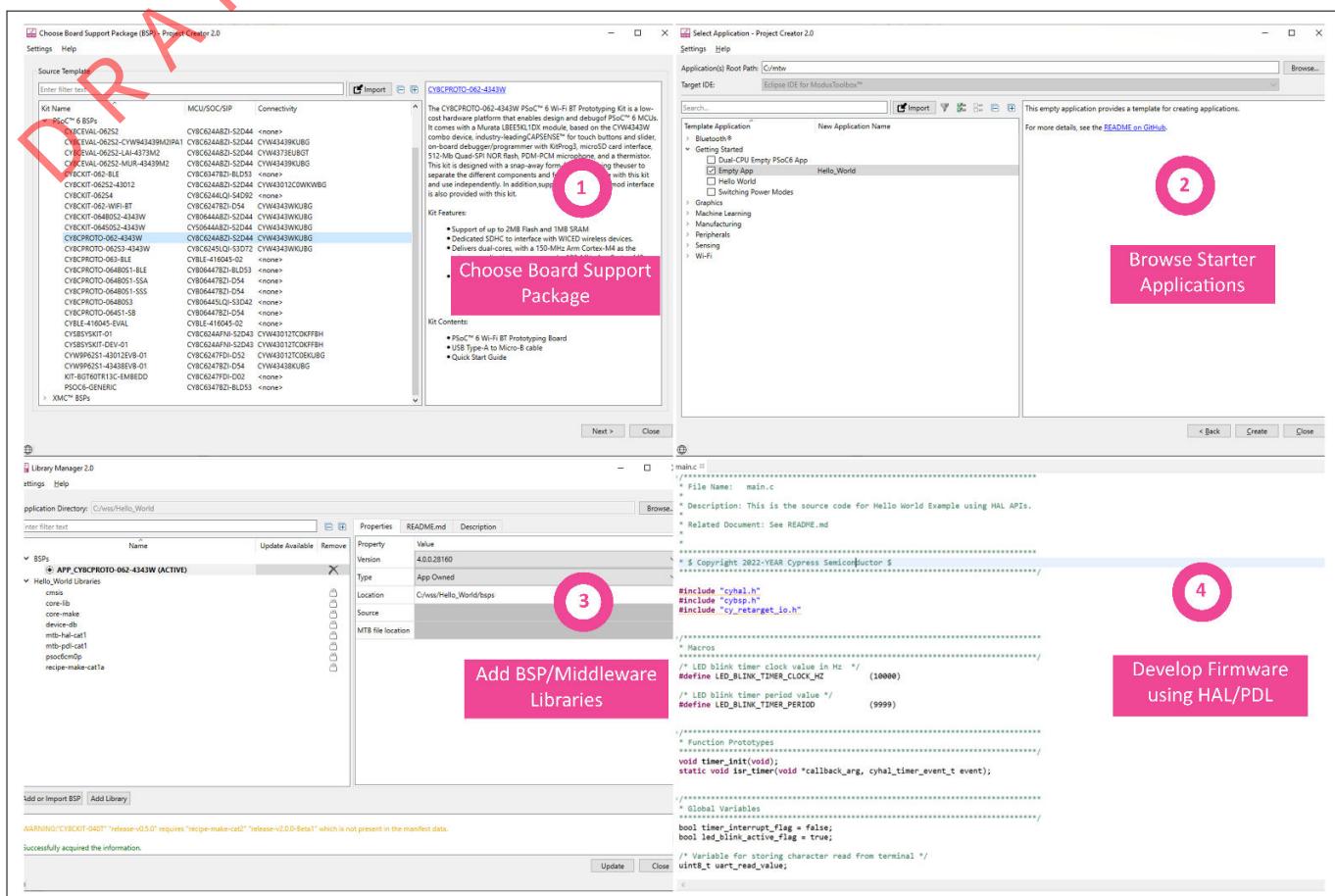


Figure 10 ModusToolbox™ resources and middleware

5.1.2.2.6 ModusToolbox™ help

The ModusToolbox™ ecosystem provides documentation and training. One way to access it is launching the Eclipse IDE for ModusToolbox™ software and navigating to the following **Help** menu items:

Choose **Help > ModusToolbox™ General Documentation**:

- ModusToolbox™ Documentation Index:** Provides brief descriptions and links to various types of documentation included as part the ModusToolbox™ software.
 - ModusToolbox™ Installation Guide:** Provides instructions for installing the ModusToolbox™ software.
 - ModusToolbox™ User Guide:** This guide primarily covers the ModusToolbox™ aspects of building, programming and debugging applications. It also covers various aspects of the tools installed along with the IDE.
 - ModusToolbox™ Training Class Material:** Links to the training material available at <https://github.com/Infineon/training-modustoolbox>.
 - Release Notes**
- For documentation on Eclipse IDE for ModusToolbox™, choose **Help > Eclipse IDE for ModusToolbox™ General Documentation**:
- Quick Start Guide:** Provides you the basics for using Eclipse IDE for ModusToolbox™
 - User Guide:** Provides descriptions about creating applications as well as building, programming, and debugging them using Eclipse IDE

5 PSoC™ 6 application notes

- **Eclipse IDE for ModusToolbox™ Help:** Provides description on how to create new applications, update application code, change middleware settings, and program/debug applications
- **Eclipse IDE Survival Guide**

5.1.2.3 Support for other IDEs

You can develop firmware for PSoC™ 6 MCUs using your favorite IDE such as [IAR Embedded Workbench](#), [Keil µVision 5](#) or [Visual Studio Code](#).

ModusToolbox™ configurators are stand-alone tools that can be used to set up and configure PSoC™ 6 MCU resources and other middleware components without using the Eclipse IDE. The Device Configurator and middleware configurators use the design.x files within the application workspace. You can then point to the generated source code and continue developing firmware in your IDE.

If there is a change in the device configuration, edit the design.x files using the configurators and regenerate the code. It is recommended that you generate resource configurations using the configuration tools provided with ModusToolbox™ software.

See [ModusToolbox™ tools package user guide](#) for details.

5.1.2.4 FreeRTOS support with ModusToolbox™

Adding native FreeRTOS support to a ModusToolbox™ application project is like adding any middleware library. You can include the FreeRTOS middleware in your application by using the Library Manager. If using the Eclipse IDE, select the application project and click the **Library Manager** link in the **Quick Panel**. Click **Add Library** and select **freertos** from the **Core** dialog, as [Figure 11](#) shows.

The .mtb file pointing to the FreeRTOS middleware is added to the application project's deps directory. The middleware content is also downloaded and placed inside the corresponding folder called **freertos**. The default location is in the shared asset repo named mtb_shared. To continue working with FreeRTOS follow the steps in the Quick Start section of [FreeRTOS documentation](#).

5 PSoC™ 6 application notes

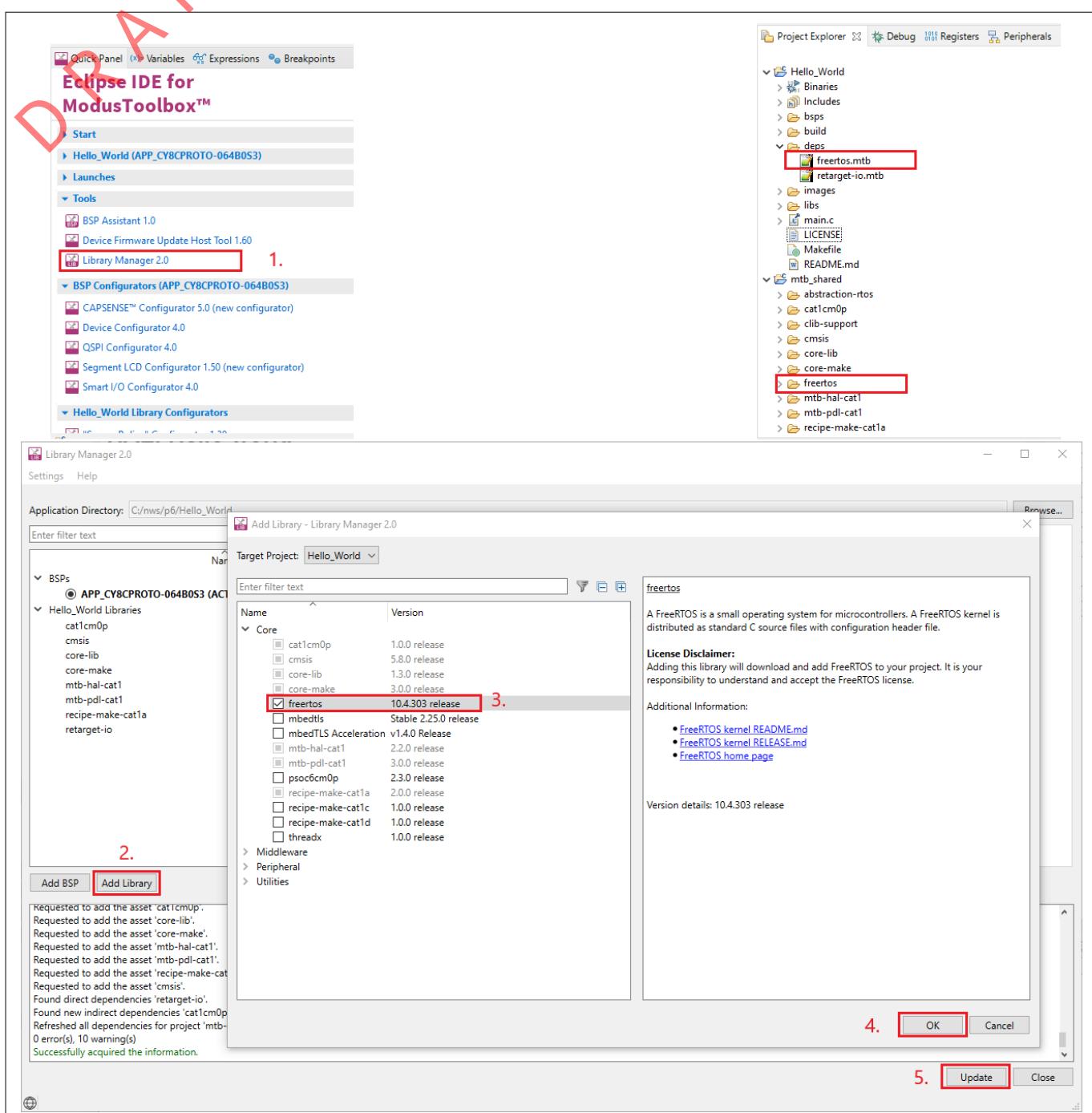


Figure 11 Import FreeRTOS middleware in ModusToolbox™ application

5.1.2.5

Programming/debugging using Eclipse IDE

All PSoC™ 6 Kits have a KitProg3 onboard programmer/debugger. It supports Cortex® Microcontroller Software Interface Standard - Debug Access Port (CMSIS-DAP). See the [KitProg3 user guide](#) for details.

The Eclipse IDE requires KitProg3 and uses the [OpenOCD](#) protocol for debugging PSoC™ 6 MCU applications. It also supports GDB debugging using industry standard probes like the [Segger J-Link](#).

Note: The [PSoC™ 6 Wi-Fi-Bluetooth® pioneer kit \(CY8CKIT-062-WiFi-BT\)](#) and [PSoC™ 6 Bluetooth® LE pioneer kit \(CY8CKIT-062-BLE\)](#) have the KitProg2 onboard programmer/debugger firmware pre-installed. To work with ModusToolbox™, upgrade the firmware to KitProg3 using the fw-loader command-line tool

5 PSoC™ 6 application notes

included in the ModusToolbox™ software. Refer to the PSoC™ 6 Programming/Debugging - KitProg Firmware Loader section in the [Eclipse IDE for ModusToolbox™ user guide](#) for more details.

For more information on debugging firmware on PSoC™ devices with ModusToolbox™, refer to the *Program and Debug* section in the [Eclipse IDE for ModusToolbox™ user guide](#).

5.1.2.6 PSoC™ 6 MCU development kits

Table 2 Development kits

Product line	Development kits
Performance	PSoC™ 6 WiFi-Bluetooth® pioneer kit (CY8CKIT-062-WiFi-BT) PSoC™ 6 Wi-Fi Bluetooth® prototyping kit (CY8CPROTO-062-4343W) PSoC™ 62S2 Wi-Fi Bluetooth® pioneer kit (CY8CKIT-062S2-43012) PSoC™ 62S3 Wi-Fi Bluetooth® prototyping kit (CY8CPROTO-062S3-4343W) PSoC™ 62S1 Wi-Fi Bluetooth® pioneer kit (CYW9P62S1-43438EVB-01) PSoC™ 62S1 Wi-Fi Bluetooth® pioneer kit (CYW9P62S1-43012EVB-01) PSoC™ 62S4 pioneer kit (CY8CKIT-062S4)
Connectivity	PSoC™ 6 Bluetooth® LE pioneer kit (CY8CKIT-062-BLE) PSoC™ 6 Bluetooth® LE prototyping kit (CY8CPROTO-063-BLE)
Security	PSoC™ 64 “Secure Boot” Wi-Fi Bluetooth® pioneer kit (CY8CKIT-064B0S2-4343W) PSoC™ 64 Standard “Secure” - AWS Wi-Fi Bluetooth® pioneer kit (CY8CKIT-064S0S2-4343W)

For the complete list of kits for the PSoC™ 6 MCU along with the shield modules, see the [Microcontroller \(MCUs\) kits](#) page.

~~DRAFT~~ 5 PSoC™ 6 application notes

5.1.3 Device features

PSoC™ 6 MCUs have extensive features as shown in [Figure 12](#). The following is a list of major features. For more information, see the device [datasheet](#), the [technical reference manual \(TRM\)](#), and the section on [References](#).

- **MCU Subsystem**

- 150-MHz Arm® Cortex®-M4 and 100-MHz Arm® Cortex®-M0+
- Up to 2 MB of flash with additional 32 KB for EEPROM emulation and 32-KB supervisory flash
- Up to 1 MB of SRAM with selectable Deep Sleep retention granularity at 32-KB retention boundaries
- Inter-processor communication supported in hardware
- DMA controllers

- **Security features**

- Cryptography accelerators and true random number generator function
- One-time programmable eFUSE for secure key storage
- “Secure Boot” with hardware hash-based authentication

- **I/O subsystem**

- Up to 104 GPIOs with programmable drive modes, drive strength, slew rates
- Two ports with Smart I/O that can implement Boolean operations

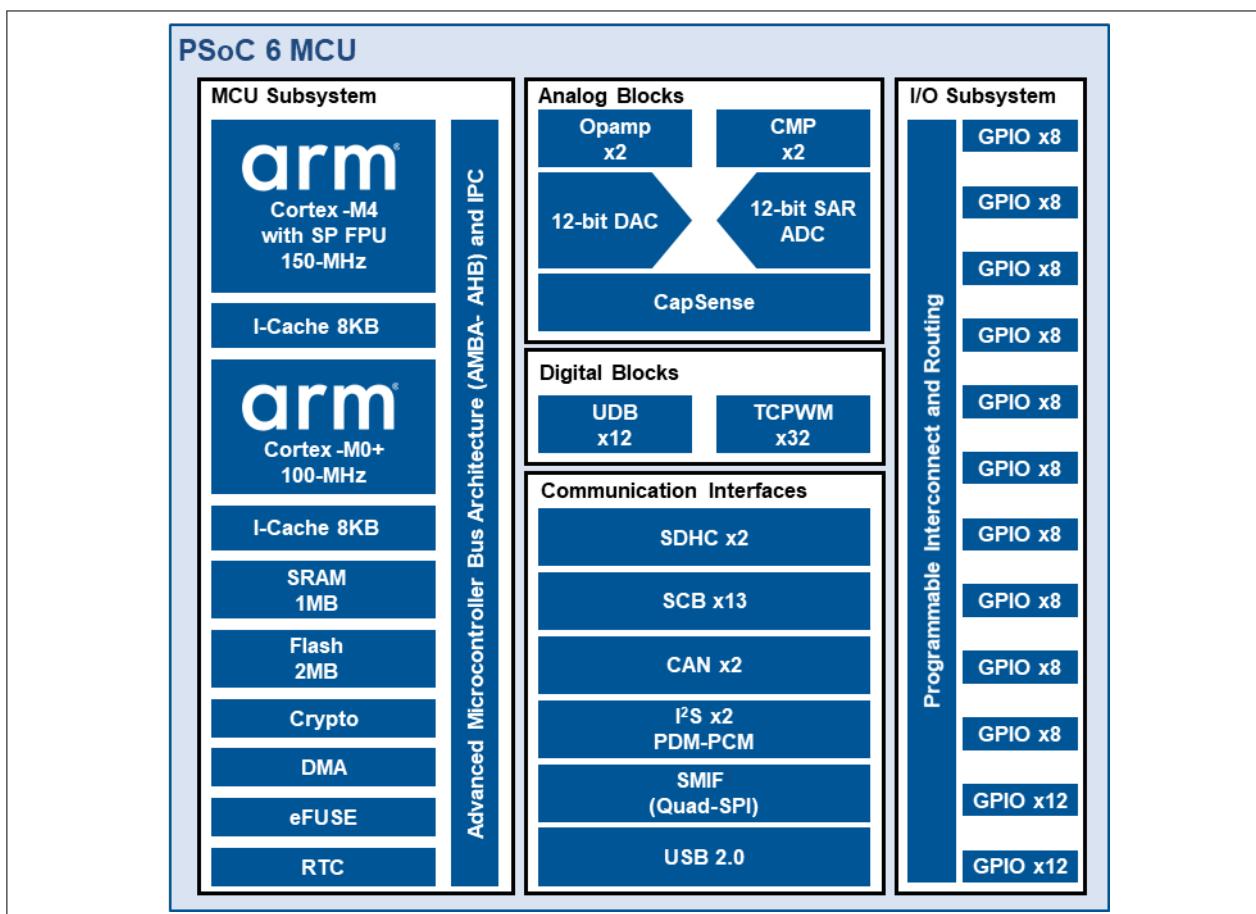


Figure 12 PSoC™ 6 MCU block diagram

- **Programmable digital blocks, communication interfaces**

- Up to 12 UDBs for custom digital peripherals
- Up to 32 TCPWM blocks configurable as 16-bit/ 32-bit timer, counter, PWM, or quadrature decoder
- Up to 13 SCBs configurable as I²C Master or Slave, SPI Master or Slave, or UART

5 PSoC™ 6 application notes

- ~~DRAFT~~
- Controller Area Network interface with Flexible Data-Rate
 - Up to two "Secure" Digital Host Controllers with support for SD, SDIO, and eMMC interfaces
 - Audio subsystem with up to two I2S interface and two PDM channels
 - SMIF interface with support for execute-in-place from external quad SPI flash memory and on-the-fly encryption and decryption
 - USB full-speed device interface
 - **Programmable analog blocks**
 - Up to two opamps that can operate in system deep sleep mode
 - Up to two 12-bit SAR ADCs with maximum of 2-Msps sample rate and capability to function in system deep sleep mode in some of the PSoC™ 6 MCUs
 - One 12-bit, 500 ksps voltage-mode DAC
 - Up to two low-power comparators which can be used to wake up the device from all the low-power modes
 - 1.2-V bandgap reference with 1% tolerance for use with SAR ADCs and the DAC
 - **CAPSENSE™ with SmartSense auto-tuning**
 - Supports both CAPSENSE™ Sigma-Delta (CSD) and CAPSENSE™ Transmit/Receive (CSX) controllers
 - Provides best-in-class SNR, liquid tolerance, and proximity sensing
 - **Operating voltage range, power domains, and low-power modes**
 - Device operating voltage: 1.71 V to 3.6 V with user-selectable core logic operation at either 1.1 V or 0.9 V
 - Multiple on-chip regulators: low-drop out (LDO for Active, Deep Sleep modes), buck converter
 - Six power modes for fine-grained power management
 - An "Always ON" backup power domain with built-in RTC, power management integrated circuit (PMIC) control, and limited SRAM backup

5 PSoC™ 6 application notes

5.1.4 My first PSoC™ 6 MCU design using Eclipse IDE for ModusToolbox™ software

This section does the following:

- Demonstrate how to build a simple PSoC™ 6 MCU-based design and program it on to the development kit
- Makes it easy to learn PSoC™ 6 MCU design techniques and how to use the Eclipse IDE for ModusToolbox™ software

5.1.4.1 Prerequisites

Before you get started, make sure that you have the appropriate development kit for your PSoC™ 6 MCU product line, and have installed the required software. You also need internet access to the GitHub repositories during project creation.

5.1.4.1.1 Hardware

The example design shown below is developed for the [PSoC™ 6 Wi-Fi Bluetooth® prototyping kit \(CY8CPROTO-062-4343W\)](#). However, you can build the application for other development kits. See the [Using these instructions](#) section.

5.1.4.1.2 Software

- [ModusToolbox™](#) 3.0 or above

After installing the software, refer to the [ModusToolbox™ tools package user guide](#) to get an overview of the software.

5.1.4.2 Using these instructions

These instructions are grouped into several sections. Each section is devoted to a phase of the application development workflow. The major sections are:

- [Part 1: Create a new application](#)
- [Part 2: View and modify the design configuration](#)
- [Part 3: Write firmware](#)
- [Part 4: Build the application](#)
- [Part 5: Program the device](#)
- [Part 6: Test your design](#)

This design is developed for the [PSoC™ 6 Wi-Fi Bluetooth® prototyping kit \(CY8CPROTO-062-4343W\)](#). You can use other supported kits to test this example by selecting the appropriate kit while creating the application. The code described in the sections that follow has been tested on the following additional kits.

- [PSoC™ 6 Wi-Fi Bluetooth® pioneer kit \(CY8CKIT-062-WiFi-BT\)](#)
- [PSoC™ 6 Bluetooth® LE pioneer kit \(CY8CKIT-062-BLE\)](#)
- [PSoC™ 6 Bluetooth® LE prototyping kit \(CY8CPROTO-063-BLE\)](#)
- [PSoC™ 62S2 Wi-Fi Bluetooth® pioneer kit \(CY8CKIT-062S2-43012\)](#)
- [PSoC™ 62S1 Wi-Fi Bluetooth® pioneer kit \(CYW9P62S1-43438EVB-01\)](#)
- [PSoC™ 62S1 Wi-Fi Bluetooth® pioneer kit \(CYW9P62S1-43012EVB-01\)](#)
- [PSoC™ 62S3 Wi-Fi Bluetooth® prototyping kit \(CY8CPROTO-062S3-4343W\)](#)
- [PSoC™ 64 “Secure Boot” Wi-Fi Bluetooth® pioneer kit \(CY8CKIT-064B0S2-4343W\)](#)
- [PSoC™ 62S4 pioneer kit \(CY8CKIT-062S4\)](#)

~~DETAILED~~ 5 PSoC™ 6 application notes

5.1.4.3 About the design

This design uses the CM4 core of the PSoC™ 6 MCU to execute two tasks: UART communication and LED control. At device reset, the Infineon-supplied pre-built CM0+ application image enables the CM4 core and configures the CM0+ core to go to sleep. The CM4 core uses the UART to print a “Hello World” message to the serial port stream, and starts blinking the user LED on the kit. When the user presses the enter key on the serial console, the blinking is paused or resumed.

5.1.4.4 Part 1: Create a new application

This section takes you on a step-by-step guided tour of the new application process. It uses the **Empty App** starter application and manually adds the functionality from the **Hello World** starter application. The Eclipse IDE for ModusToolbox™ is used in the instructions, but you can use any IDE or the command line if you prefer.

If you are familiar with developing projects with ModusToolbox™ software, you can use the **Hello World** starter application directly. It is a complete design, with all the firmware written for the supported kits. You can walk through the instructions and observe how the steps are implemented in the code example.

If you start from scratch and follow all the instructions in this application note, you can use the **Hello World** code example as a reference while following the instructions.

Launch the Eclipse IDE for ModusToolbox™ to get started. Please note that ModusToolbox™ software needs access to the internet to successfully clone the starter application onto your machine.

1. Select a new workspace.

At launch, Eclipse IDE for ModusToolbox™ presents a dialog to choose a directory for use as the workspace directory. The workspace directory is used to store workspace preferences and development artifacts. You can choose an existing empty directory by clicking the **Browse** button, as [Figure 13](#) shows. Alternatively, you can type in a directory name to be used as the workspace directory along with the complete path, and the Eclipse IDE will create the directory for you.

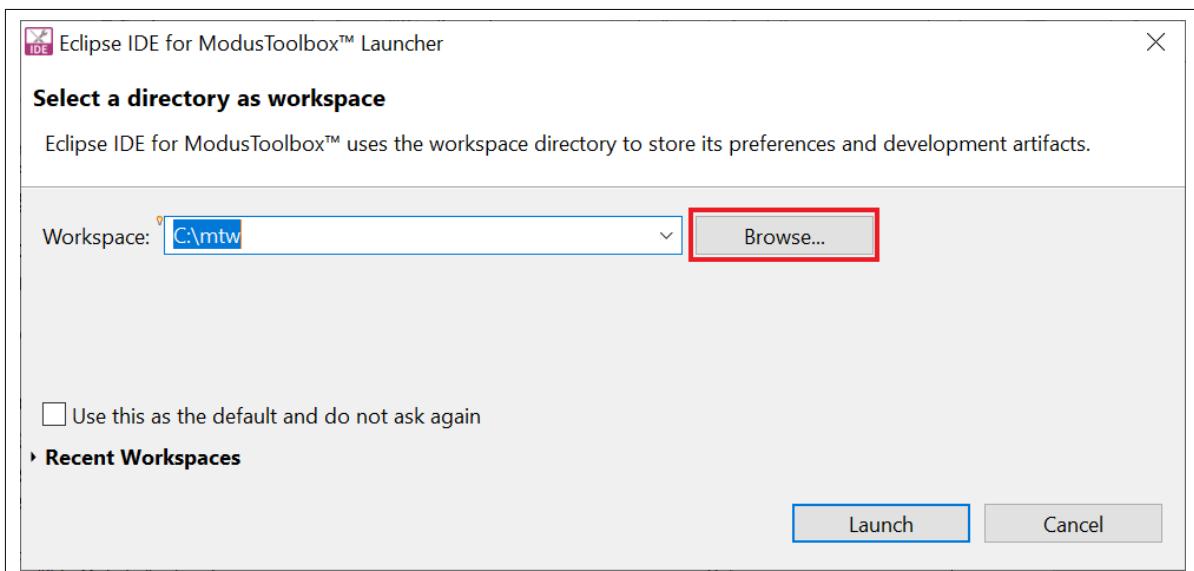


Figure 13 Select a directory as the workspace

2. Create a new ModusToolbox™ application.

- a. Click **New Application** in the Start group of the Quick Panel.
- b. Alternatively, you can choose **File > New > ModusToolbox™ Application**, as [Figure 14](#) shows. The Project Creator opens.

5 PSoC™ 6 application notes

DRAFT

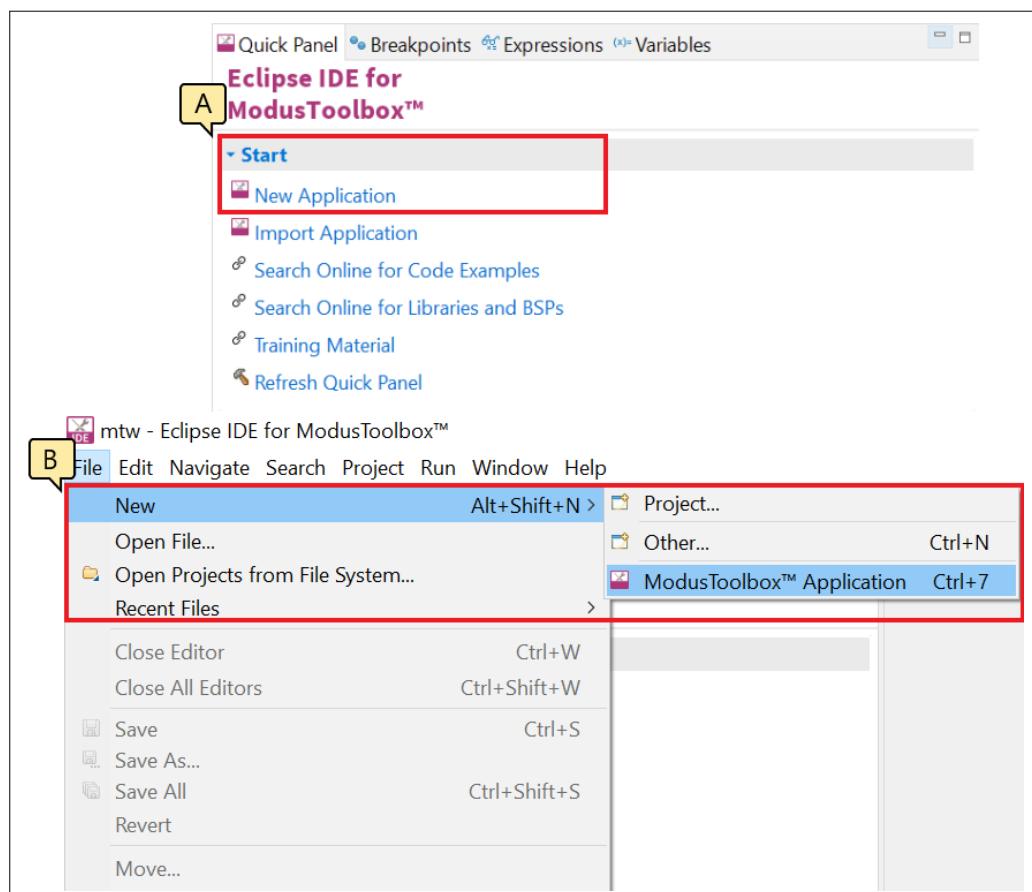


Figure 14 Create a new ModusToolbox™ application

3. Select a target PSoC™ 6 development kit

ModusToolbox™ speeds up the development process by providing BSPs that set various workspace/project options for the specified development kit in the new application dialog.

- In the **Choose Board Support Package (BSP)** dialog, choose the **Kit Name** that you have. The steps that follow use **CY8CPROTO-062-4343W**. See [Figure 15](#) for help with this step
- Click **Next**

5 PSoC™ 6 application notes

DRAFT

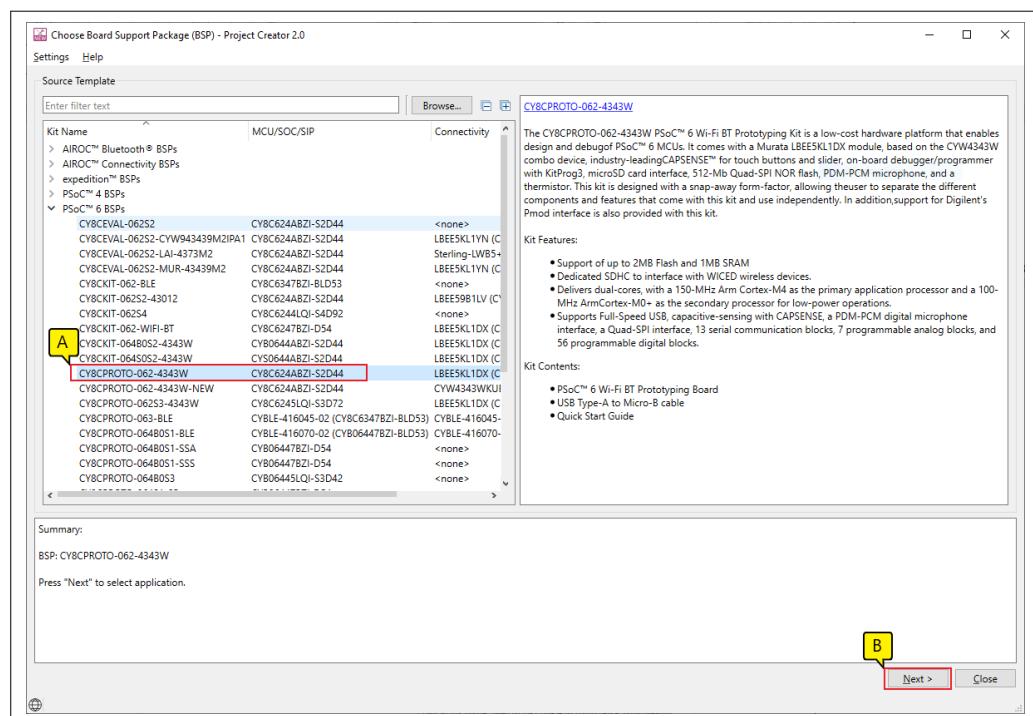


Figure 15 Choose target hardware

- c. In the **Select Application** dialog, select **Empty App** starter application, as [Figure 16](#) shows.
- d. In the **Name** field, type in a name for the application, such as **Hello_World**. You can choose to leave the default name if you prefer.
- e. Click **Create** to create the application, as [Figure 16](#) shows, wait for the Project Creator to automatically close once the project is successfully created.

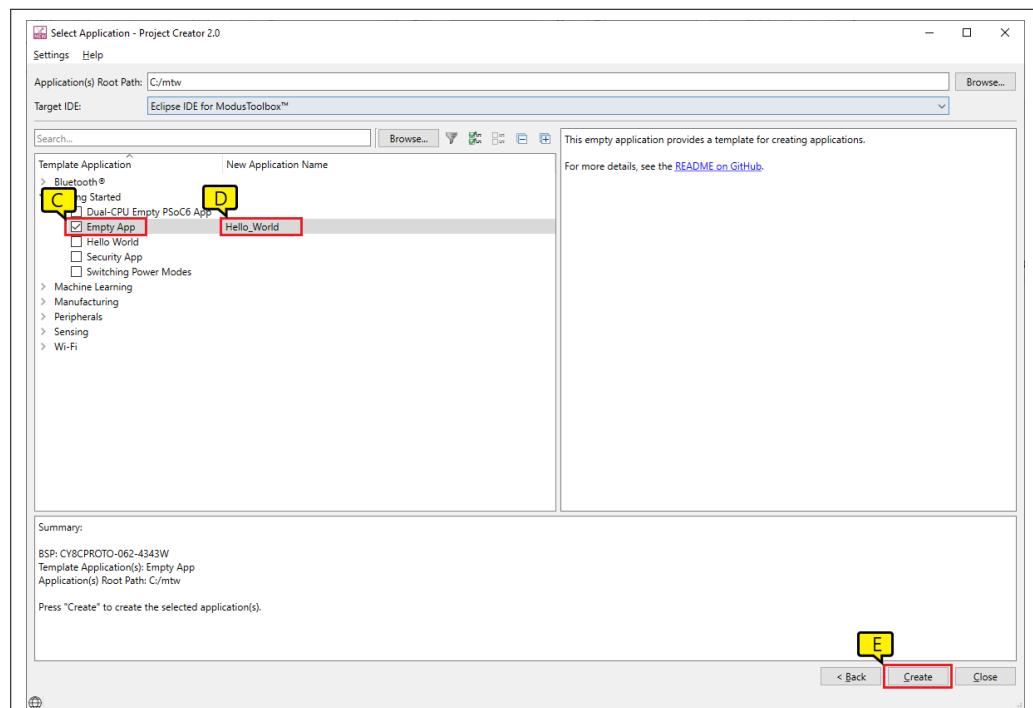


Figure 16 Choose starter application

You have successfully created a new ModusToolbox™ application for a PSoC™ 6 MCU.

The BSP uses CY8C624ABZI-D54 as the default device that is mounted on the [PSoC™ 6 Wi-Fi-Bluetooth® prototyping kit \(CY8CPROTO-062-4343W\)](#) along with the CYW4343WKUBG Wi-Fi/Bluetooth® radio.

5 PSoC™ 6 application notes

If you are using custom hardware based on PSoC™ 6 MCU, or a different PSoC™ 6 MCU part number, please refer to the Custom BSP App Note or the BSP Assistant user guide.

5.1.4.5 Part 2: View and modify the design configuration

Figure 17 shows the Eclipse IDE Project Explorer interface displaying the structure of the application project.

A PSoC™ 6 MCU application consists of a project to develop code for the CM4 core. A project folder consists of various subfolders – each denoting a specific aspect of the project.

5 PSoC™ 6 application notes

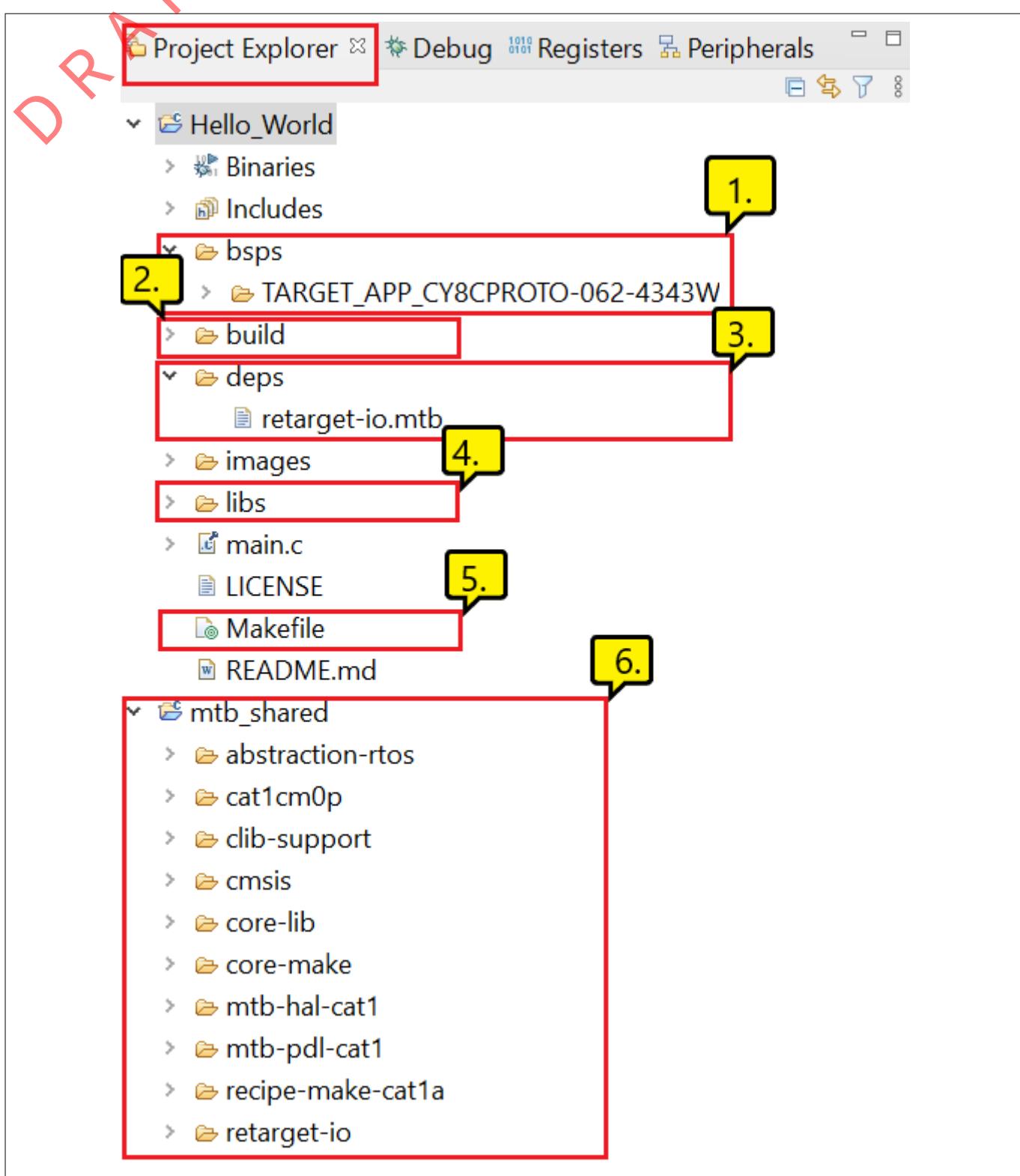


Figure 17 Project Explorer view

1. The files provided by the BSP are in the **bsps** folder and are listed under **TARGET_<bsp name>** sub-folders. All the input files for the device and peripheral configurators are in the **config** folder inside the BSP. The **GeneratedSource** folder in the BSP contains the files that are generated by the configurators and are prefixed with **cycfg_**. These files contain the design configuration as defined by the BSP. From ModusToolbox™ 3.x or later, you can directly customize configurator files of BSP for your application

~~5 PSoC™ 6 application notes~~

rather than overriding the default design configurator files with custom design configurator files since BSPs are completely owned by the application.

The BSP folder also contains the linker scripts and the start-up code for the PSoC™ 6 MCU used on the board.

- ~~2.~~ The build folder contains all the artifacts resulting from a build of the project. The output files are organized by target BSPs.
- ~~3.~~ The deps folder contains .mtb files, which provide the locations from which ModusToolbox™ pulls the libraries that are directly referenced by the application. These files typically each contain the GitHub location of a library. The .mtb files also contain a git Commit Hash or Tag that tells which version of the library is to be fetched and a path as to where the library should be stored locally.

For example, Here, retarget-io.mtb points to mtb://retarget-io#latest-v1.X##\$ASSET_REPO\$\$/retarget-io/latest-v1.x. The variable \$\$ASSET_REPO\$\$ points to the root of the shared location which defaults to mtb_shared. If the library must be local to the application instead of shared, use \$\$LOCAL\$\$ instead of \$\$ASSET_REPO\$\$.

- ~~4.~~ The libs folder also contains .mtb files. In this case, they point to libraries that are included indirectly as a dependency of a BSP or another library. For each indirect dependency, the Library Manager places an .mtb file in this folder. These files have been populated based on the targets available in deps folder.
- For example, using BSP cy8cproto-062-4343W populates libs folder with the following .mtb files: *cmsis.mtb*, *core-lib.mtb*, *core-make.mtb*, *mtb-hal-cat1.mtb*, *mtb-pdl-cat1.mtb*, *cat1cm0p.mtb*, *recipe-make-cat1a.mtb*.

The libs folder contains the file mtb.mk, which stores the relative paths of all the libraries required by the application. The build system uses this file to find all the libraries required by the application.

Everything in the libs folder is generated by the Library Manager so you should not manually edit anything in that folder.

- ~~5.~~ An application contains a Makefile which is at the application's root folder. This file contains the set of directives that the make tool uses to compile and link the application project. There can be more than one project in an application. In that case there is a Makefile at the application level and one inside each project. See [AN215656 - PSoC™ 6 MCU dual-core system design](#) for details related to multi-project applications.
 - ~~6.~~ By default, when creating a new application or adding a library to an existing application and specifying it as shared, all libraries are placed in an mtb_shared directory adjacent to the application directories.
- The mtb_shared folder is shared between different applications within a workspace. Different applications may use different versions of shared libraries if necessary.

5.1.4.5.1 Opening the Device Configurator

BSP configurator files are in the bsp/TARGET_<BSP-name>/config folder. For example, click <Application-name> from **Project Explorer** then click **Device Configurator** link in the **Quick Panel** to open the file design.modus in the **Device Configurator** as shown in [Figure 18](#). You can also open other configuration files in their respective configurators or click the corresponding links in the **Quick Panel**.

5 PSoC™ 6 application notes

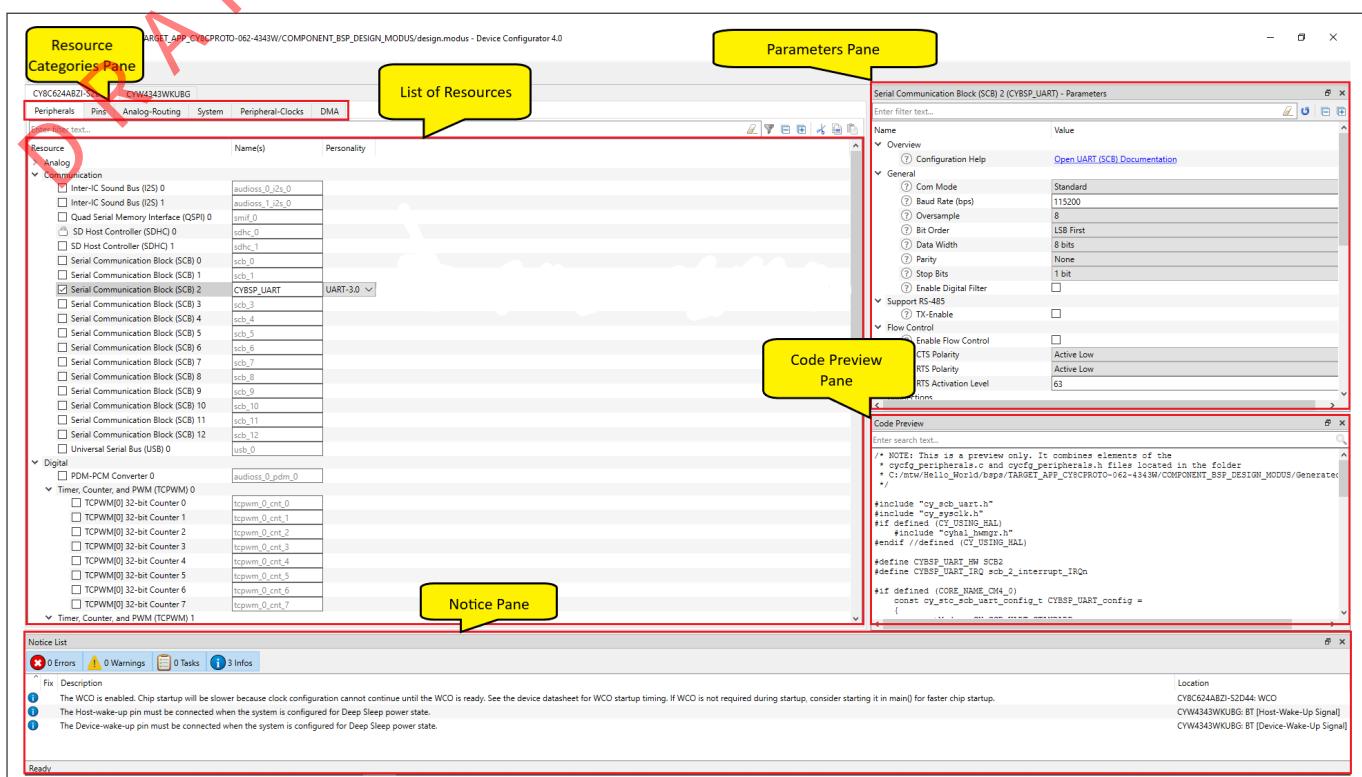


Figure 18 Device Configurator

The **DeviceConfigurator** provides a set of **Resources Categories** tabs. Here you can choose between different resources available in the device such as peripherals, pins, and clocks from the **List of Resources**.

You can choose how a resource behaves by choosing a **Personality** for the resource. For example, a **Serial Communication Block (SCB)** resource can have **EZI2C**, **I2C**, **SPI**, or **UART** personalities. The **Alias** is your name for the resource, which is used in firmware development. One or more aliases can be specified by using a comma to separate them (with no spaces).

The **Parameters** pane is where you enter the configuration parameters for each enabled resource and the selected personality. The **Code Preview** pane shows the configuration code generated per the configuration parameters selected. This code is populated in the `cycfg_` files in the `GeneratedSource` folder. The Parameters pane and Code Preview pane may be displayed as tabs instead of separate windows but the contents will be the same.

Any errors, warnings, and information messages arising out of the configuration are displayed in the Notices pane.

Currently, the device configurator supports configurations using PDL source. If you choose to use HAL libraries in your application then you do not need to do any device configurations changes in here. The application project contains source files that help you create an application for the CM4 core (for example, `main.c`), while the CM0+ application is supplied as a default c file (`psoc6_02_cm0p_sleep.c` for the CY8C624ABZI-D44 device). See the [cat1cm0p](#) library. This c file is compiled and linked with the CM4 image as part of the normal build process.

At this point in the development process, we are ready to add the required middleware to the design. The only middleware required for the Hello World application is the [retarget-io](#) library.

5.1.4.5.2 Add retarget-io middleware

In this step, you will add the [retarget-io](#) middleware to redirect standard input and output streams to the UART configured by the BSP. The initialization of the middleware will be done in `main.c`.

1. In the **Quick Panel**, click the **Library Manager** link.

5 PSoC™ 6 application notes

- ~~DO NOT USE~~
2. In the subsequent dialog, click **Add Libraries**.
 3. Under **Peripherals**, select and enable **retarget-io**.
 4. Click **OK** and then **Update**.

The files necessary to use the **retarget-io** middleware are added in the `mtb_shared > retarget_io` folder, and the `.mtb` file is added to the `deps` folder, as [Figure 19](#) shows.

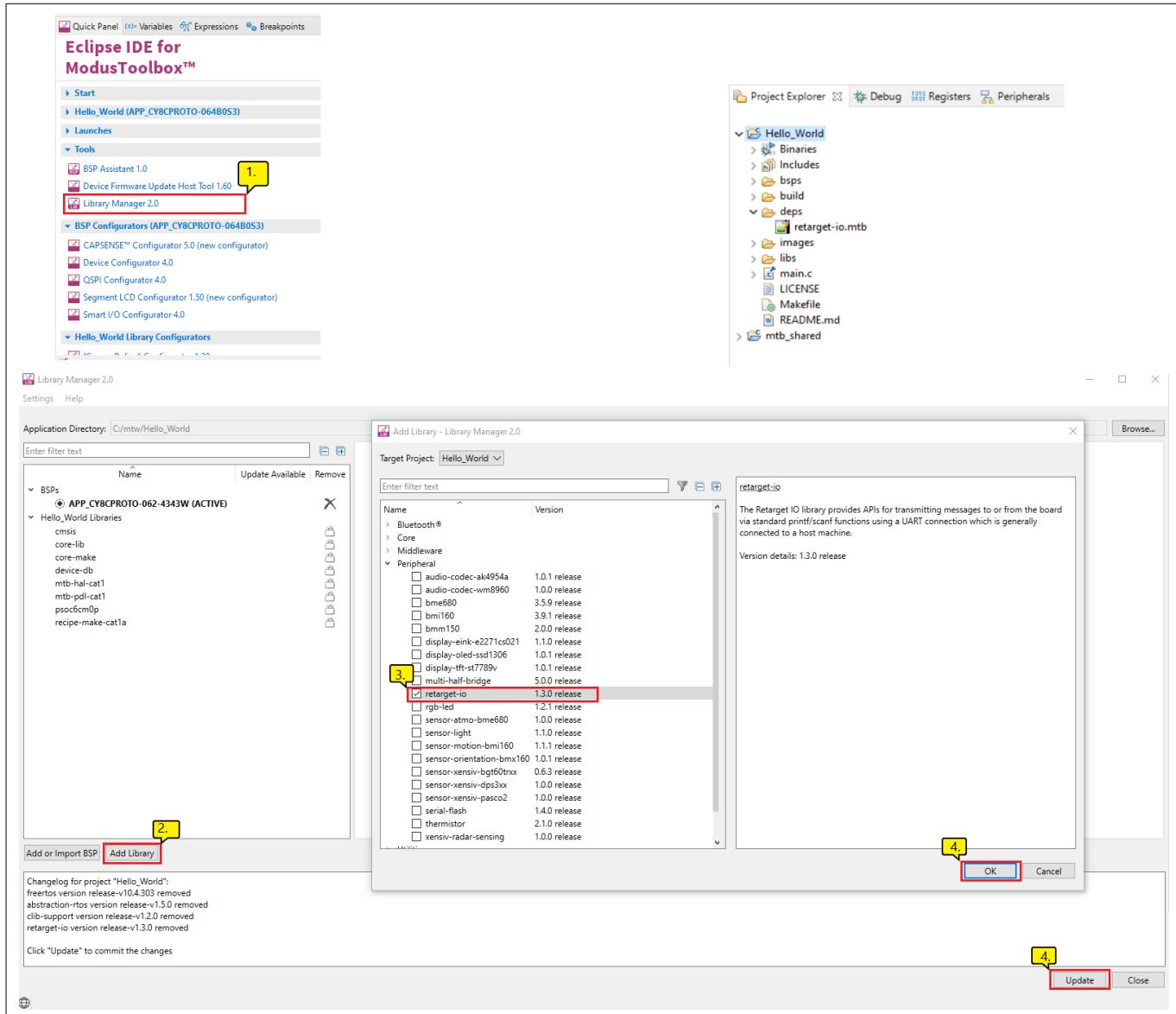


Figure 19 Add the **retarget-io** middleware

5.1.4.5.3 Configuration of UART, timer peripherals, pins, and system clocks

The configuration of the debug UART peripheral, timer peripheral, pins and system clocks can be done directly in the code using the function APIs provided by the BSP and HAL. Therefore, it is not necessary to configure them with the Device Configurator. See [Part 3: Write firmware](#).

5 PSoC™ 6 application notes~~DRAFT~~
5.1.4.6 Part 3: Write firmware

At this point in the development process, you have created an application with the assistance of an application template and modified it to add the [retarget-io](#) middleware. In this part, you write the firmware that implements the design functionality.

If you are working from scratch using the Empty PSoC™ 6 starter application, you can copy the respective source code to the *main.c* of the application project from the code snippet provided in this section. If you are using the Hello World code example, all the required files are already in the application.

Firmware flow

We now examine the code in the *main.c* file of the application. [Figure 20](#) shows the firmware flowchart.

The CM0+ core comes out of reset and enables the CM4 core. The CM0+ core is then configured to go to sleep by the provided CM0+ application. Resource initialization for this example is performed by the CM4 core. It configures the system clocks, pins, clock to peripheral connections, and other platform resources.

When the CM4 core is enabled, the clocks and system resources are initialized by the BSP initialization function. The [retarget-io](#) middleware is configured to use the debug UART, and the user LED is initialized. The debug UART prints a "Hello World!" message on the terminal emulator – the on-board KitProg3 acts the USB-UART bridge to create the virtual COM port. A timer object is configured to generate an interrupt every 1000 milliseconds. At each Timer interrupt, the CM4 core toggles the LED state on the kit.

The firmware is designed to accept the 'Enter' key as an input and on every press of the 'Enter' key the firmware starts or stops the blinking of the LED.

Note that the application code uses BSP/HAL/middleware functions to execute the intended functionality.

`cybsp_init()`- This BSP function sets up the HAL hardware manager and initializes all the system resources of the device including but not limited to the system clocks and power regulators.

`cy_retarget_io_init()`- This function from the [retarget-io](#) middleware uses the aliases set up in the BSP for the debug UART pins to configure the debug UART with a standard baud rate of 115200 and also redirects the input/output stream to the debug UART.

Note: You can open the Device Configurator to view the aliases that are set up in the BSP.

`cyhal_gpio_init()`- This function from the GPIO HAL initializes the physical pin to drive the LED. The LED used is derived from the alias for the pin set up in the BSP.

`timer_init()`- This function wraps a set of timer HAL function calls to instantiate and configure a hardware timer. It also sets up a callback for the timer interrupt.

Copy the following code snippet to *main.c* of your application project.

5 PSoC™ 6 application notes

Code listing 1: main.c file

```
#include "cyhal.h"
#include "cybsp.h"
#include "cy_retarget_io.h"

/****************************************************************************
 * Macros
 **************************************************************************/

/* LED blink timer clock value in Hz */
#define LED_BLINK_TIMER_CLOCK_HZ          (10000)

/* LED blink timer period value */
#define LED_BLINK_TIMER_PERIOD           (9999)

/****************************************************************************
 * Function Prototypes
 **************************************************************************/

void timer_init(void);
static void isr_timer(void *callback_arg, cyhal_timer_event_t event);

/****************************************************************************
 * Global Variables
 **************************************************************************/

bool timer_interrupt_flag = false;
bool led_blink_active_flag = true;

/* Variable for storing character read from terminal */
uint8_t uart_read_value;

/* Timer object used for blinking the LED */
cyhal_timer_t led_blink_timer;

/****************************************************************************
 * Function Name: main
 **************************************************************************/

* Summary:
* This is the main function for CM4 core. It sets up a timer to trigger a
* periodic interrupt. The main while loop checks for the status of a flag set
* by the interrupt and toggles an LED at 1Hz to create an LED blinky. The
* while loop also checks whether the 'Enter' key was pressed and
* stops/restarts LED blinking.
*
* Parameters:
* none
*
* Return:
* int
```

5 PSoC™ 6 application notes

DRAFT

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

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

    /* Board init failed. Stop program execution */
    if (result != CY_RSLT_SUCCESS)
    {
        CY_ASSERT(0);
    }

    /* Enable global interrupts */
    __enable_irq();

    /* Initialize retarget-io to use the debug UART port */
    result = cy_retarget_io_init(CYBSP_DEBUG_UART_TX, CYBSP_DEBUG_UART_RX,
                                CY_RETARGET_IO_BAUDRATE);

    /* retarget-io init failed. Stop program execution */
    if (result != CY_RSLT_SUCCESS)
    {
        CY_ASSERT(0);
    }

    /* Initialize the User LED */
    result = cyhal_gpio_init(CYBSP_USER_LED, CYHAL_GPIO_DIR_OUTPUT,
                            CYHAL_GPIO_DRIVE_STRONG, CYBSP_LED_STATE_OFF);

    /* GPIO init failed. Stop program execution */
    if (result != CY_RSLT_SUCCESS)
    {
        CY_ASSERT(0);
    }

    /* \x1b[2J\x1b[;H - ANSI ESC sequence for clear screen */
    printf("\x1b[2J\x1b[;H");

    printf("***** "
          "PSoC 6 MCU: Hello World! Example "
          "***** \r\n\r\n");

    printf("Hello World!!!\r\n\r\n");

    printf("For more PSoC 6 MCU projects, "
          "visit our code examples repositories:\r\n\r\n");

    printf("https://github.com/Infineon/"
          "Code-Examples-for-ModusToolbox-Software\r\n\r\n");
}
```

5 PSoC™ 6 application notes

DRAFT

```

/* Initialize timer to toggle the LED */
timer_init();

printf("Press 'Enter' key to pause or "
      "resume blinking the user LED \r\n\r\n");

for (;;)
{
    /* Check if 'Enter' key was pressed */
    if (cyhal_uart_getc(&cy_retarget_io_uart_obj, &uart_read_value, 1)
        == CY_RSLT_SUCCESS)
    {
        if (uart_read_value == '\r')
        {
            /* Pause LED blinking by stopping the timer */
            if (led_blink_active_flag)
            {
                cyhal_timer_stop(&led_blink_timer);

                printf("LED blinking paused \r\n");
            }
            else /* Resume LED blinking by starting the timer */
            {
                cyhal_timer_start(&led_blink_timer);

                printf("LED blinking resumed\r\n");
            }

            /* Move cursor to previous line */
            printf("\x1b[1F");

            led_blink_active_flag ^= 1;
        }
    }

    /* Check if timer elapsed (interrupt fired) and toggle the LED */
    if (timer_interrupt_flag)
    {
        /* Clear the flag */
        timer_interrupt_flag = false;

        /* Invert the USER LED state */
        cyhal_gpio_toggle(CYBSP_USER_LED);
    }
}

*****  

* Function Name: timer_init  

*****  

* Summary:  

* This function creates and configures a Timer object. The timer ticks

```

5 PSoC™ 6 application notes

~~DRAFT~~

```
* continuously and produces a periodic interrupt on every terminal count
* event. The period is defined by the 'period' and 'compare_value' of the
* timer configuration structure 'led_blink_timer_cfg'. Without any changes,
* this application is designed to produce an interrupt every 1 second.
*
* Parameters:
* none
*
*****void timer_init(void)
{
    cy_rslt_t result;

    const cyhal_timer_cfg_t led_blink_timer_cfg =
    {
        .compare_value = 0,                      /* Timer compare value, not used */
        .period = LED_BLINK_TIMER_PERIOD,        /* Defines the timer period */
        .direction = CYHAL_TIMER_DIR_UP,         /* Timer counts up */
        .is_compare = false,                     /* Don't use compare mode */
        .is_continuous = true,                  /* Run timer indefinitely */
        .value = 0                               /* Initial value of counter */
    };

    /* Initialize the timer object. Does not use input pin ('pin' is NC) and
     * does not use a pre-configured clock source ('clk' is NULL). */
    result = cyhal_timer_init(&led_blink_timer, NC, NULL);

    /* timer init failed. Stop program execution */
    if (result != CY_RSLT_SUCCESS)
    {
        CY_ASSERT(0);
    }

    /* Configure timer period and operation mode such as count direction,
     * duration */
    cyhal_timer_configure(&led_blink_timer, &led_blink_timer_cfg);

    /* Set the frequency of timer's clock source */
    cyhal_timer_set_frequency(&led_blink_timer, LED_BLINK_TIMER_CLOCK_HZ);

    /* Assign the ISR to execute on timer interrupt */
    cyhal_timer_register_callback(&led_blink_timer, isr_timer, NULL);

    /* Set the event on which timer interrupt occurs and enable it */
    cyhal_timer_enable_event(&led_blink_timer, CYHAL_TIMER_IRQ_TERMINAL_COUNT,
                           7, true);

    /* Start the timer with the configured settings */
    cyhal_timer_start(&led_blink_timer);
}

*****
```

5 PSoC™ 6 application notes

DRAFT

```
* Function Name: isr_timer
*****
* Summary:
* This is the interrupt handler function for the timer interrupt.
*
* Parameters:
*   callback_arg    Arguments passed to the interrupt callback
*   event           Timer/counter interrupt triggers
*
*****
static void isr_timer(void *callback_arg, cyhal_timer_event_t event)
{
    (void) callback_arg;
    (void) event;

    /* Set the interrupt flag and process it from the main while(1) loop */
    timer_interrupt_flag = true;
}
```

5 PSoC™ 6 application notes

DRAFT

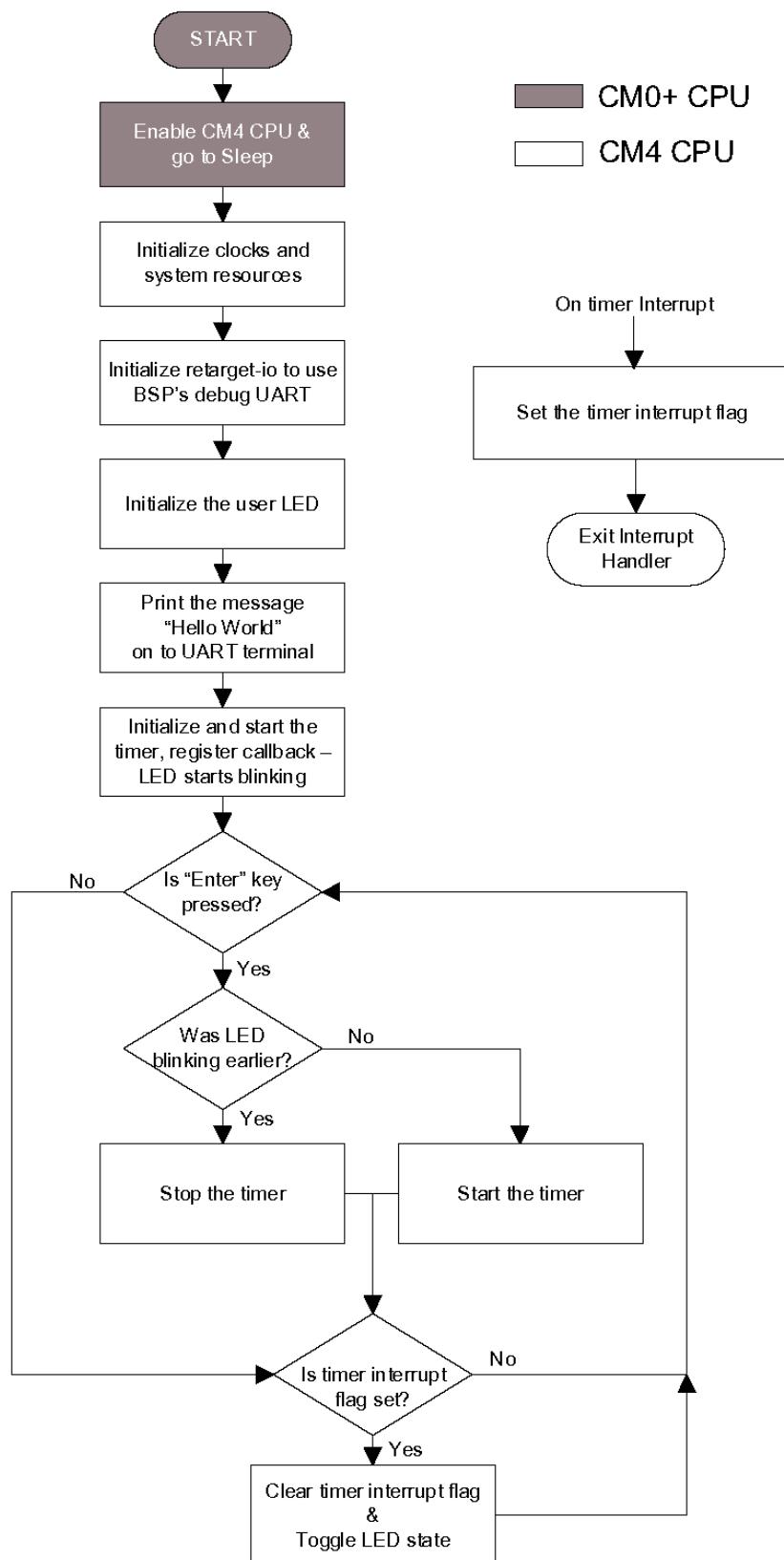


Figure 20 Firmware flowchart

This completes the summary of how the firmware works in the code example. Feel free to explore the source files for a deeper understanding.

5 PSoC™ 6 application notes

5.1.4.7 Part 4: Build the application

This section shows how to build the application.

1. Select the application project in the **Project Explorer** view.

2. Click **Build Application** shortcut under the <name> group in the **Quick Panel**.

It selects the build configuration from Makefile and compiles/links all projects that constitute the application. By default, Debug configurations are selected.

3. The **Console view** lists the results of the build operation, as [Figure 21](#) shows.

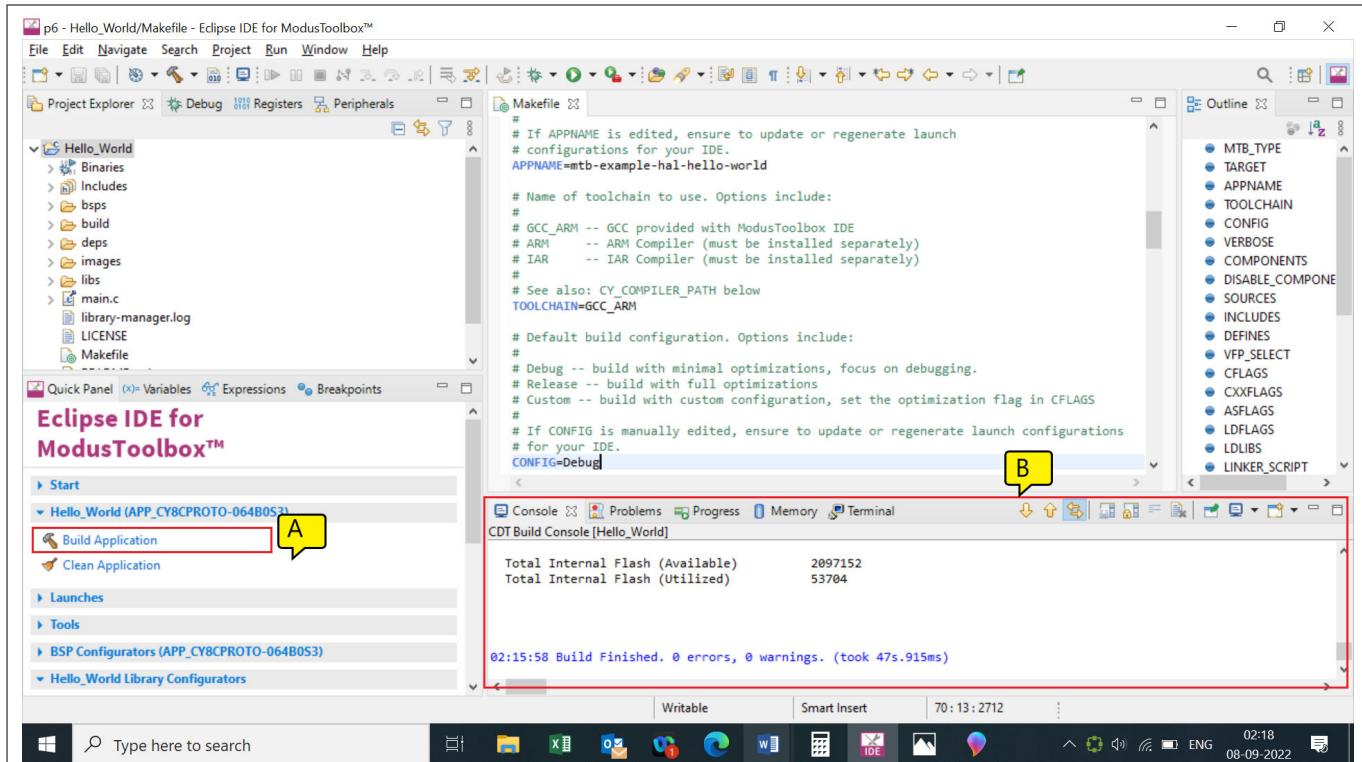


Figure 21 Build the application

If you encounter errors, revisit prior steps to ensure that you completed all the required tasks.

Note: You can also use the command line interface (CLI) to build the application. Please refer to the **Build system chapter** in the [ModusToolbox™ tools package user guide](#). This document is located in the /docs_<version>/ folder in the ModusToolbox™ installation.

5.1.4.8 Part 5: Program the device

This section shows how to program the PSoC™ 6 MCU.

ModusToolbox™ software uses the OpenOCD protocol to program and debug applications on PSoC™ 6 MCUs. The kit must be running KitProg3. Some kits are shipped with KitProg2 firmware instead of KitProg3. See [Programming/debugging using Eclipse IDE](#) for details. The ModusToolbox™ tools package includes the fw-loader command-line tool to switch the KitProg firmware from KitProg2 to KitProg3. See the PSoC™ 6 MCU KitProg Firmware Loader section in the [Eclipse IDE for ModusToolbox™ user guide](#) for more details.

If you are using a development kit with a built-in programmer (the CY8CPROTO-062-4343W, for example), connect the board to your computer using the USB cable.

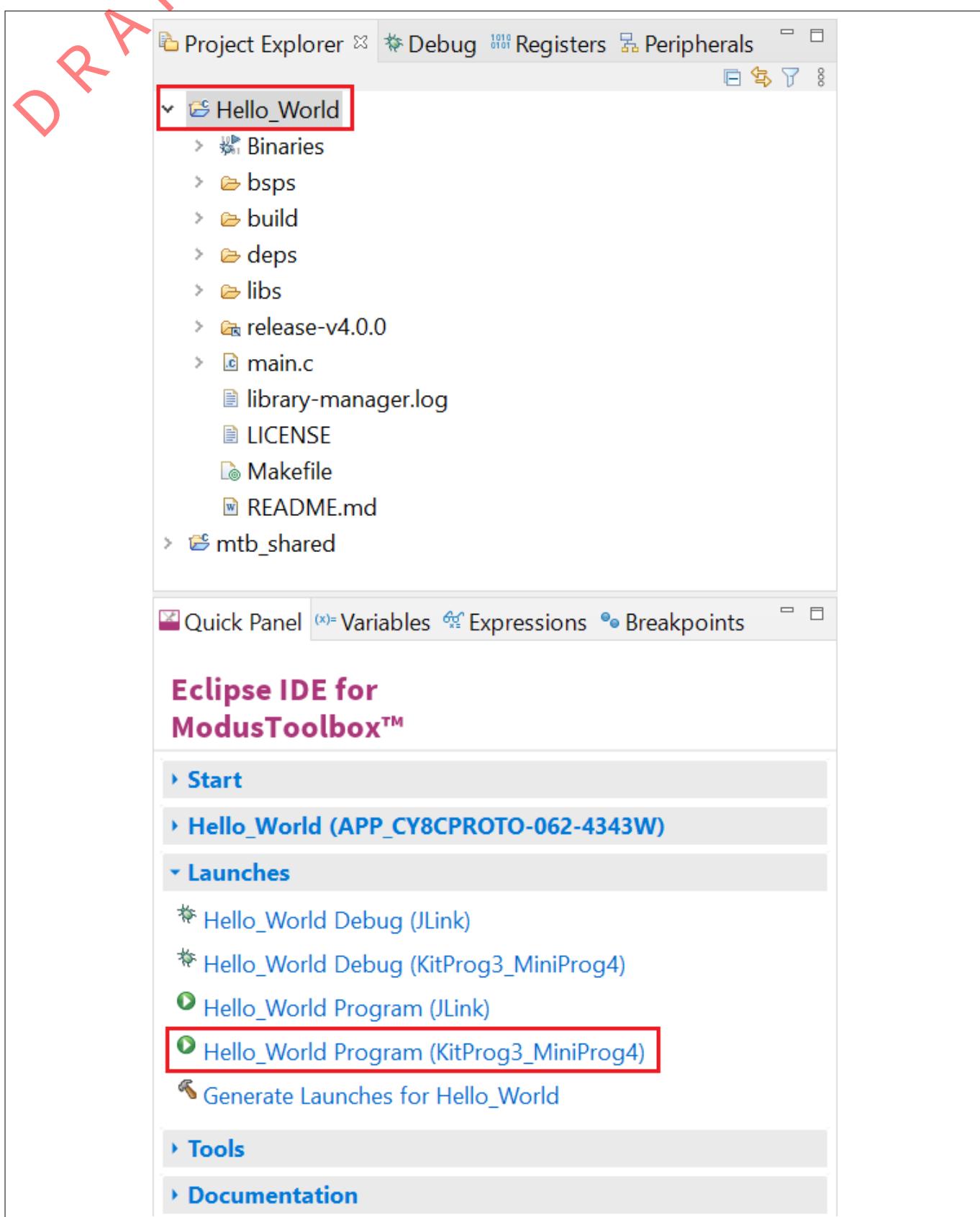
If you are developing on your own hardware, you can use a hardware programmer/debugger; for example, a [CY8CKIT-005 MiniProg4](#), [J-Link](#), or [ULINKpro](#).

5 PSoC™ 6 application notes

Select the application project and click the <application name> Program (KitProg3_MiniProg4) shortcut under the **Launches** group in the **Quick Panel**, as [Figure 22](#) shows. The IDE will select and run the appropriate run configuration.

Note: *This step also performs a build if any files have been modified since the last build.*

5 PSoC™ 6 application notes

**Figure 22 Programming an application to a device**

The Console view lists the results of the programming operation, as Figure 23 shows.

5 PSoC™ 6 application notes

```

<terminated> Hello_World Program [KitProg3] [GDB OpenOCD Debugging] openocd.exe
[ 79% ][#####][ Erasing ] [ 82% ][#####][ Erasing ] [ 85% ][#####][ Erasing ] [ 89% ][#####][ Erasing ] [ 91% ][#####][ Erasing ] [ 94% ][#####][ Erasing ] [ 97% ][#####][ Erasing ] [ 100% ][#####][ Erasing ]
[ 78% ][#####][ Programming ] [ 79% ][#####][ Programming ] [ 83% ][#####][ Programming ] [ 89% ][#####][ Programming ] [ 95% ][#####][ Programming ] [ 100% ][#####][ Programming ]
write 44832 bytes from file C:/mtw/Hello_World/build/CY8CPROTO-062-4343W/Debug/mbt-example-psoc6-empty-app.hex in 1.873642s (22.950 KiB/s)
** Programming finished **
** Program operation completed successfully **
srst_only separate srst_gates_jtag srst_open_drain connect_deassert_srst
Info : SWD DPIDR 0xb0a02477
shutdown command invoked

```

Figure 23 Programming an application to a device

5.1.4.9 Part 6: Test your design

This section describes how to test your design.

Follow the steps below to observe the output of your design. This note uses Tera Term as the UART terminal emulator to view the results. You can use any terminal of your choice to view the output.

1. Select the serial port

Launch Tera Term and select the USB-UART COM port as [Figure 24](#) shows. Note that your COM port number may be different.

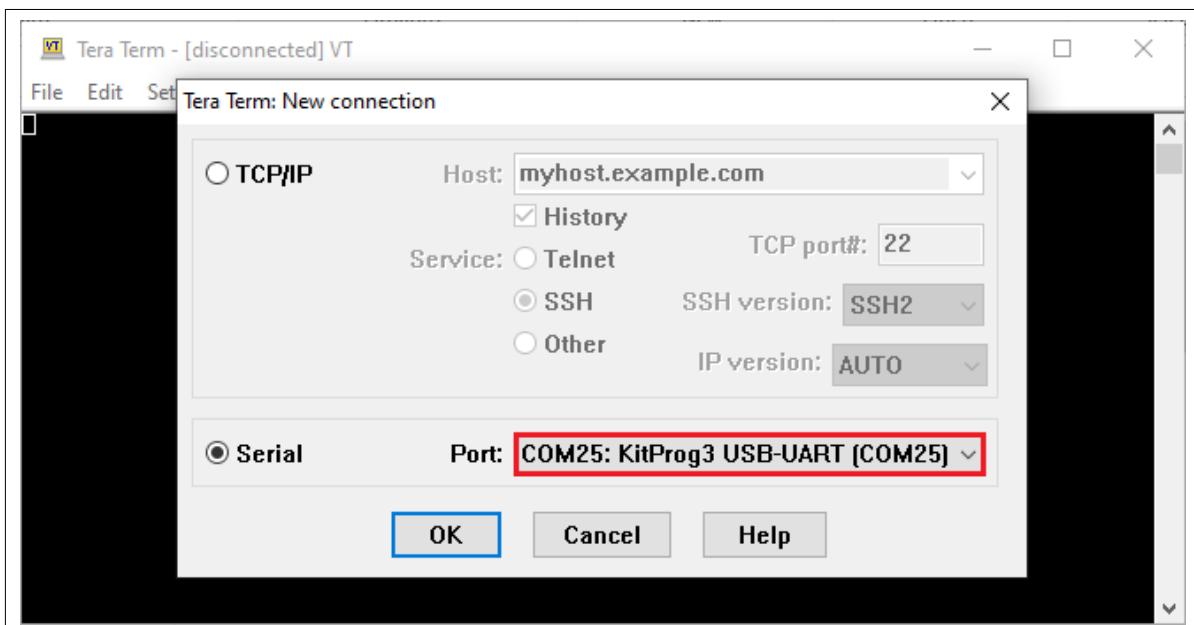


Figure 24 Selecting the KitProg3 COM port in Tera Term

2. Set the baud rate

Set the baud rate to 115200 under **Setup > Serial port** as [Figure 25](#) shows.

5 PSoC™ 6 application notes

DRAFT

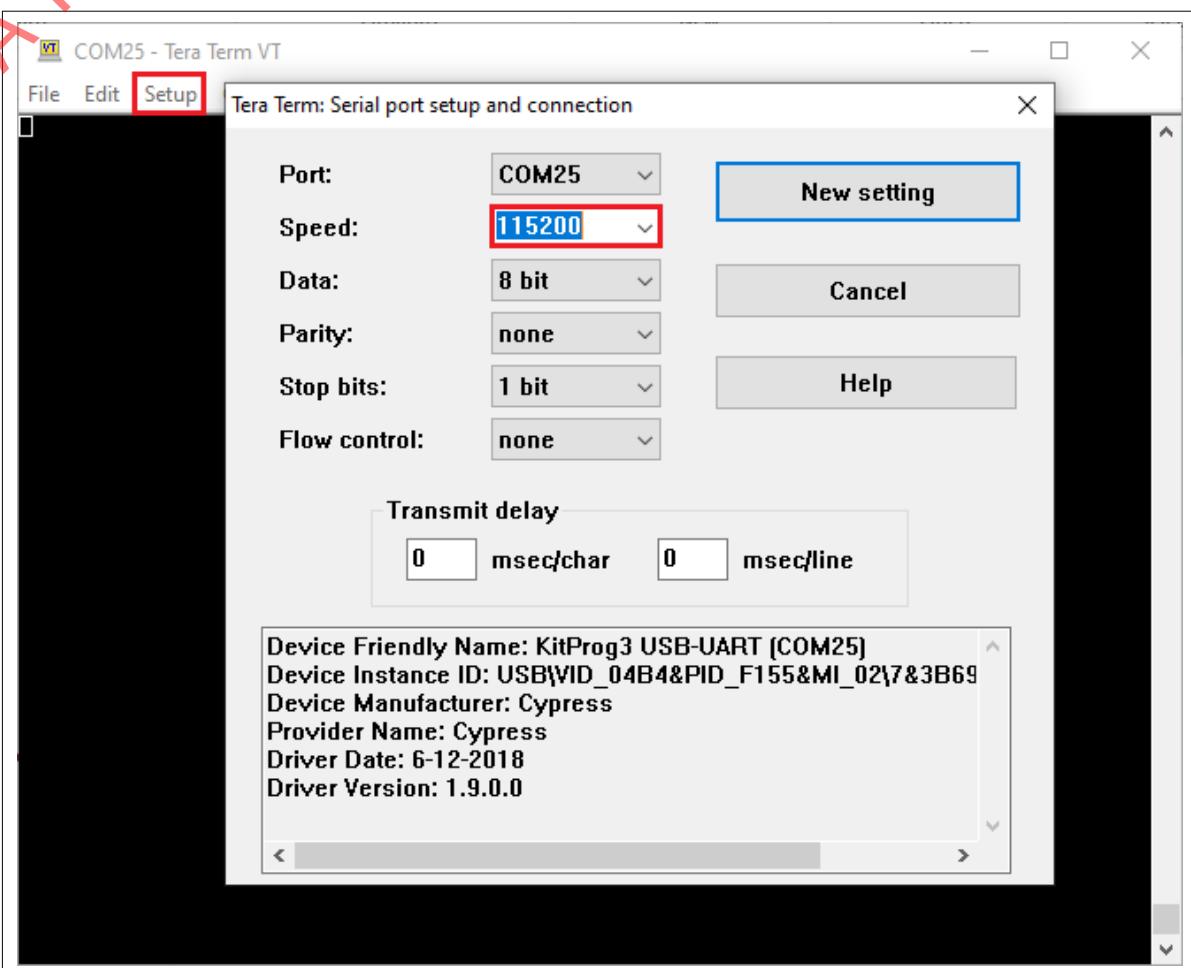


Figure 25

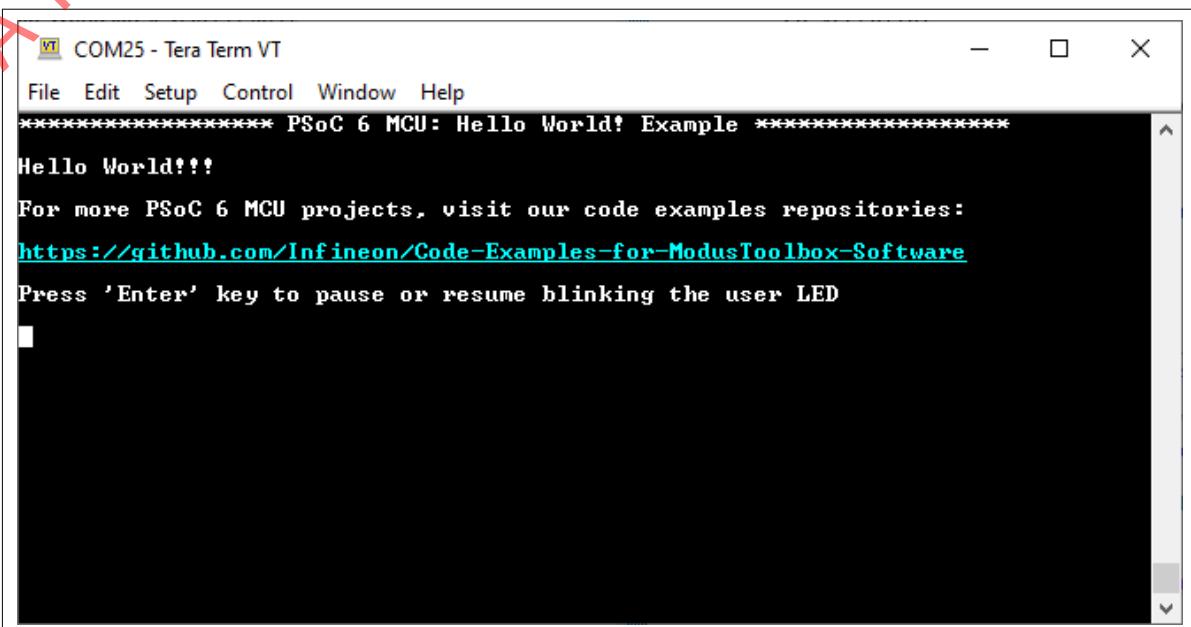
Configuring the baud rate in Tera Term

3. Reset the device

Press the reset switch (**SW1**) on the kit. A message appears on the terminal as [Figure 26](#) shows. The user LED on the kit will start blinking.

5 PSoC™ 6 application notes

DRAFT



COM25 - Tera Term VT

File Edit Setup Control Window Help

***** PSoC 6 MCU: Hello World! Example *****

Hello World!!!

For more PSoC 6 MCU projects, visit our code examples repositories:

<https://github.com/Infineon/Code-Examples-for-ModusToolbox-Software>

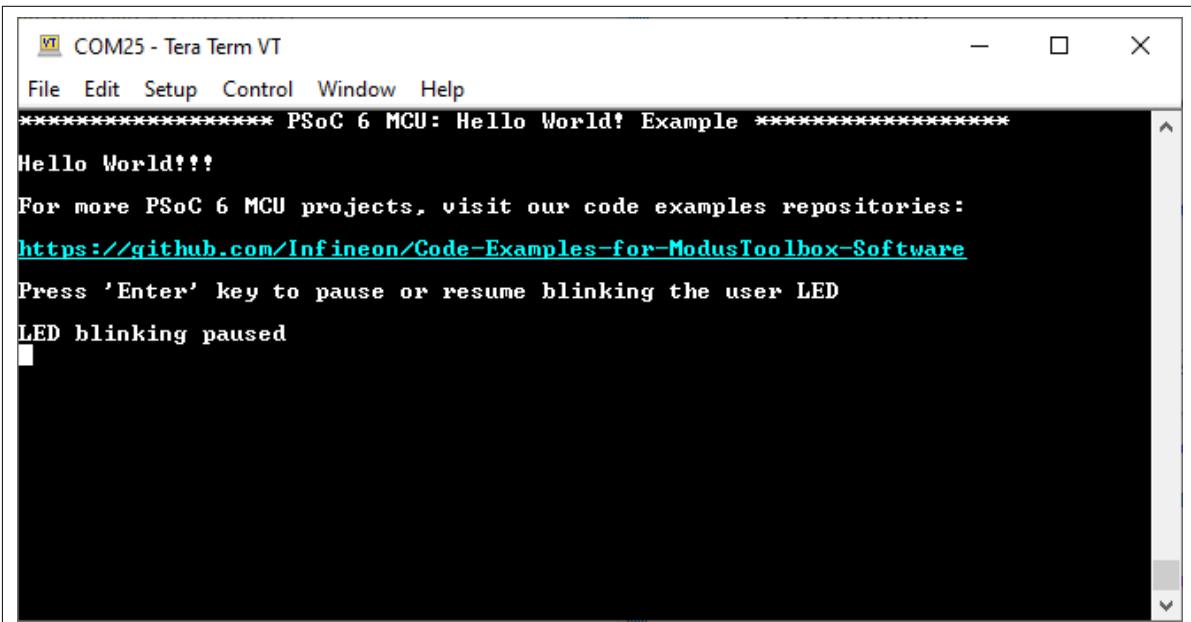
Press 'Enter' key to pause or resume blinking the user LED

█

Figure 26 UART message printed from CM4 core

4. Pause/resume LED blinking functionality

Press the **Enter** key to pause/resume blinking the LED. When the LED blinking is paused, a corresponding message will be displayed on the terminal as Figure 27 shows.



COM25 - Tera Term VT

File Edit Setup Control Window Help

***** PSoC 6 MCU: Hello World! Example *****

Hello World!!!

For more PSoC 6 MCU projects, visit our code examples repositories:

<https://github.com/Infineon/Code-Examples-for-ModusToolbox-Software>

Press 'Enter' key to pause or resume blinking the user LED

LED blinking paused

█

Figure 27 UART message from CM4 core

5 PSoC™ 6 application notes~~DRAFT~~
5.1.5 Summary

This application note explored the PSoC™ 6 MCU device architecture and the associated development tools. PSoC™ 6 MCU is a truly programmable embedded system-on-chip with configurable analog and digital peripheral functions, memory, and a dual-core system on a single chip. The integrated features and low-power modes make PSoC™ 6 MCU an ideal choice for smart home, IoT gateways, and other related applications.

5 PSoC™ 6 application notes

References

-
- 1

For a complete and updated list of PSoC™ 6 MCU code examples, please visit our [GitHub](#). For more PSoC™ 6 MCU-related documents, please visit our [PSoC™ 6 MCU product web page](#).

Table 3 lists the system-level and general application notes that are recommended for the next steps in learning about PSoC™ 6 MCU and ModusToolbox™.

Table 3 General and system-level application notes

Document	Document name
AN221774	Getting started with PSoC™ 6 MCU on PSoC™ Creator
AN210781	Getting started with PSoC™ 6 MCU with Bluetooth® Low Energy (BLE) connectivity on PSoC™ Creator
AN218241	PSoC™ 6 MCU hardware design considerations
AN219528	PSoC™ 6 MCU low-power modes and power reduction techniques

Table 4 lists the application notes (AN) for specific peripherals and applications.

Table 4 Documents related to PSoC™ 6 MCU features

Document	Document name
System resources, CPU, and interrupts	
AN215656	PSoC™ 6 MCU dual-core system design
AN217666	PSoC™ 6 MCU interrupts
AN235279	Performing ETM and ITM Trace on PSoC 6 MCU
CAPSENSE™	
AN92239	Proximity sensing with CAPSENSE™
AN85951	PSoC™ 4 and PSoC™ 6 MCU CAPSENSE™ design guide
Device Firmware Update	
AN213924	PSoC™ 6 MCU device firmware update software development kit guide
Low-power analog	
AN230938	PSoC™ 6 MCU low-power analog

5 PSoC™ 6 application notes

Glossary

1

This section lists the most commonly used terms that you might encounter while working with PSoC™ family of devices.

- **Board support package (BSP):** A BSP is the layer of firmware containing board-specific drivers and other functions. The board support package is a set of libraries that provide firmware APIs to initialize the board and provide access to board level peripherals.
- **Cypress Programmer:** Cypress Programmer is a flexible, cross-platform application for programming Cypress devices. It can Program, Erase, Verify, and Read the flash of the target device.
- **Hardware abstraction layer (HAL):** The HAL wraps the lower level drivers (like [MTB-PDL-CAT1](#)) and provides a high-level interface to the MCU. The interface is abstracted to work on any MCU.
- **KitProg:** The KitProg is an onboard programmer/debugger with USB-I2C and USB-UART bridge functionality. The KitProg is integrated onto most PSoC™ development kits.
- **MiniProg3/MiniProg4:** Programming hardware for development that is used to program PSoC™ devices on your custom board or PSoC™ development kits that do not support a built-in programmer.
- **Personality:** A personality expresses the configurability of a resource for a functionality. For example, the SCB resource can be configured to be an UART, SPI or I2C personalities.
- **PSoC™:** A programmable, embedded design platform that includes a CPU, such as the 32-bit Arm® Cortex®-M0, with both analog and digital programmable blocks. It accelerates embedded system design with reliable, easy-to-use solutions, such as touch sensing, and enables low-power designs.
- **Middleware:** Middleware is a set of firmware modules that provide specific capabilities to an application. Some middleware may provide network protocols (e.g. MQTT), and some may provide high level software interfaces to device features (e.g. USB, audio).
- **ModusToolbox™:** An Eclipse based embedded design platform for IoT designers that provides a single, coherent, and familiar design experience combining the industry's most deployed Wi-Fi and Bluetooth® technologies, and the lowest power, most flexible MCUs with best-in-class sensing.
- **Peripheral driver library (PDL):** The peripheral driver library (PDL) simplifies software development for the PSoC™ 6 MCU architecture. The PDL reduces the need to understand register usage and bit structures, thus easing software development for the extensive set of peripherals available.
- **WICED:** WICED (Wireless Internet Connectivity for Embedded Devices) is a full-featured platform with proven Software Development Kits (SDKs) and turnkey hardware solutions from partners to readily enable Wi-Fi and Bluetooth® connectivity in system design.

5 PSoC™ 6 application notes

5.1.6 Revision history

Document version	Date of release	Description of changes
**	2017-07-26	New application note
*A	2018-01-09	Updated screenshots with latest release of ModusToolbox™ Added new supported PSoC™ 6 MCU devices Added AnyCloud description under ModusToolbox™ software
*B	2019-04-16	Added new supported PSoC™ 6 MCU device – PSoC™ 62S4 Added information on PSoC™ 6 product lines and development kits available for each product line
*C	2020-05-06	Updated Figure 3 Updated Screenshots with MTB v2.2 Added mtb_shared folder description, updated application creation process with MTB flow
*D	2021-03-11	Updated to Infineon template
*E	2021-07-09	Updated the GitHub links Added reference to new PSoC™ 6 MCU low-power analog Updated Figure 24 to Figure 27 Firmware updated to the latest version
*F	2022-07-21	Template update
*G	2022-09-12	Updated the development flow as per MTB v3.0 software Updated link references Updated configurator info Added new Figure 2 and Figure 7 Updated Figure 8 and Figure 14 Added reference to new AN235279 Added a new section for ModusToolbox™ applications Removed reference to deprecated AN225588 Updated Figure 17 to Figure 19

5.2 AN221774 Getting started with PSoC™ 6 MCU on PSoC™ Creator

About this document

•
2

Scope and purpose

AN221774 introduces the PSoC™ 6 MCU, a dual-CPU programmable system-on-chip with Arm® Cortex® M4 and Cortex-M0+ processors. This application note helps you explore PSoC™ 6 MCU architecture and development tools, and shows you how to create your first project using PSoC™ Creator. This application note also guides you to more resources available online to accelerate your learning about PSoC™ 6 MCU. To get started with the PSoC™ 6 MCU with Bluetooth® Low Energy (LE) Connectivity device family, refer to [AN210781](#) – Getting Started with PSoC™ 6 MCU with BLE Connectivity on PSoC™ Creator.

Associated part family

All PSoC™ 6 MCU devices

5 PSoC™ 6 application notes

~~Software version~~

~~PSoC™ Creator 4.2~~

More code examples? We heard you.

To access an ever-growing list of hundreds of PSoC™ code examples, please visit our [code examples web page](#). Please visit the [GitHub](#) site for a comprehensive collection of code examples using ModusToolbox IDE for PSoC™ 6. You can also explore the PSoC™ video library [here](#).

~~DEAF~~ 5 PSoC™ 6 application notes

5.2.1 ~~DEAF~~ Introduction

PSoC™ 6 MCU is an ultra-low-power PSoC™ device with a dual-CPU architecture tailored for smart homes, IoT gateways, and so on. The PSoC™ 6 MCU device is a programmable embedded system-on-chip that integrates the following features on a single chip:

- Single-CPU microcontroller: Arm® Cortex®-M4 (CM4) or Dual-CPU microcontroller: Arm® Cortex®-M4 (CM4) and Cortex®-M0+ (CM0+)
- Programmable analog and digital peripherals
- Up to 2 MB of flash and 1 MB of SRAM
- Fourth-generation CapSense® technology
- PSoC™ 6 MCU is suitable for a variety of power-sensitive applications such as the following:
 - Smart home sensors and controllers
 - Smart home appliances
 - Gaming controllers
 - Sports, smart phone, and virtual reality (VR) accessories
 - Industrial sensor nodes
 - Industrial logic controllers
 - Advanced remote controllers

The programmable analog and digital subsystems allow flexibility and dynamic fine-tuning of the design using [PSoC™ Creator](#), the schematic-based design tool.

Figure 28 illustrates an application-level block diagram for a real-world use case using PSoC™ 6 MCU.

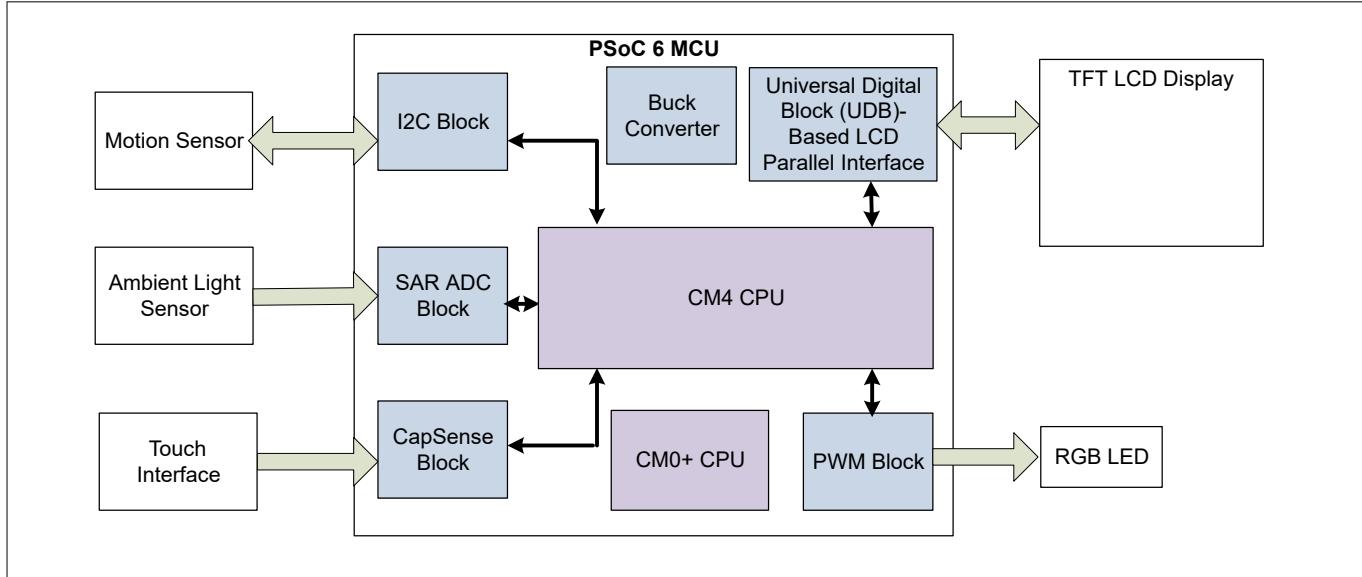


Figure 28 Application-level block diagram using PSoC™ 6 MCU

PSoC™ 6 MCU is a highly capable and flexible solution. For example, the real-world use case in [Figure 28](#) takes advantage of these features:

- A buck converter for ultra-low-power operation
- An analog front end (AFE) within the device to condition and measure sensor outputs such as ambient light sensor
- Serial Communication Blocks (SCBs) to interface with multiple digital sensors such as motion sensors
- CAPSENSE™ technology for reliable touch and proximity sensing

5 PSoC™ 6 application notes

- Digital logic (Universal Digital Blocks or UDBs) and peripherals (Timer Counter PWM or TCPWM) to drive the display and LEDs respectively.
- Product security features managed by CM0+ CPU and application features executed by CM4 CPU

See [Device features](#) and the device datasheets for more details.

This application note introduces you to the capabilities of PSoC™ 6 MCU, gives an overview of the development ecosystem, and gets you started with a simple design wherein you learn to use PSoC™ 6 MCU. This design is available as code example [CE221773](#) for PSoC™ Creator.

For hardware design considerations, see [AN218241 – PSoC™ 6 MCU Hardware Design Considerations](#).

5.2.1.1 Prerequisites

Before you get started, make sure that you have a development kit and have installed the required software. It is recommended that you download the code example for reference.

5.2.1.1.1 Hardware

- [CY8CKIT-062-WiFi-BT PSoC™ 6 Wi-Fi-BT Pioneer Kit](#)
- [CY8CKIT-062-BLE PSoC™ 6 BLE Pioneer Kit](#) or
- [CY8CPROTO-062-4343W PSoC™ 6 Wi-Fi BT Prototyping Kit](#) (not supported in PSoC™ Creator)

5.2.1.1.2 Software

- [PSoC™ Creator 4.2 with Peripheral Driver Library](#) (PDL v3.1.x or later)
- [CE221773 – PSoC™ 6 MCU Hello World Example](#)

5 PSoC™ 6 application notes

5.2.2 Development ecosystem

5.2.2.1 PSoC™ resources

A wealth of data available [here](#) helps you to select the right PSoC™ device and quickly and effectively integrate it into your design. For a comprehensive list of PSoC™ 6 MCU resources, see [How to Design with PSoC™ 6 MCU - KBA223067](#). The following is an abbreviated list of resources for PSoC™ 6 MCU.

- Overview: [PSoC™ Portfolio](#), [PSoC™ Roadmap](#)
- Product Selectors: [PSoC™ 6 MCU](#)
- [Datasheets](#) describe and provide electrical specifications for each device family
- [Application Notes](#) and [Code Examples](#) cover a broad range of topics, from basic to advanced level. You can also browse our collection of code examples. See [Code Examples](#)
- [Technical Reference Manuals \(TRMs\)](#) provide detailed descriptions of the architecture and registers in each device family
- [PSoC™ 6 MCU Programming Specification](#) provides the information necessary to program the nonvolatile memory of PSoC™ 6 MCU devices
- [CAPSENSE™ Design Guides](#): Learn how to design capacitive touch-sensing applications with PSoC™ devices
- Development Tools
 - [CY8CKIT-062-WiFi-BT PSoC™ 6 Wi-Fi-BT Pioneer Kit](#) is a development kit that supports the PSoC™ 62 series MCU along with Wi-Fi and BT connectivity
 - [CY8CKIT-062-BLE PSoC™ 6 BLE Pioneer Kit](#) is an easy-to-use and inexpensive development platform for PSoC™ 63 series MCU with BLE Connectivity
 - [CY8CPROTO-062-4343W PSoC™ 6 Wi-Fi BT Prototyping Kit](#) is a development kit that supports the PSoC™ 62 series MCU along with CYW4343W module-based Wi-Fi and Bluetooth® connectivity for development on ModusToolbox™
- [Training Videos](#): Video training on our products and tools, including a dedicated series on [PSoC™ 6 MCU](#) is available

5.2.2.2 Firmware/application development

There are two development platforms that you can use for application development with PSoC™ 6 MCU:

- **ModusToolbox™**: ModusToolbox™ software includes configuration tools, low-level drivers, middleware libraries, and operating system support, as well as other packages that enable you to create MCU and wireless applications. It also includes the optional ModusToolbox™ IDE
ModusToolbox™ IDE is an Eclipse-based development environment that runs on Windows, macOS, and Linux platforms and includes various tools
ModusToolbox™ supports stand-alone device and middleware configurators that are fully integrated into the IDE. Use the configurators to set the configuration of different blocks in the device and generate code that can be used in firmware development. ModusToolbox™ supports all PSoC™ 6 MCU devices. It is recommended that you use ModusToolbox™ for all application development for PSoC™ 6 MCUs. See the [ModusToolbox™ Software Overview](#) for more information

Libraries and enablement software are available at the [GitHub](#) site. Some resources will be used by all developers. Others will be used by developers in particular ecosystems

Software resources available at GitHub support one or more of the target ecosystems:

- MCU and Bluetooth® SoC ecosystem – a full-featured platform for PSoC™ 6, Wi-Fi, Bluetooth®, and Bluetooth® Low Energy application development

~~5 PSoC™ 6 application notes~~

- Mbed OS ecosystem – provides an embedded operating system, transport security and cloud services to create connected embedded solutions
 - Amazon FreeRTOS ecosystem – extends the FreeRTOS kernel with software libraries that make it easy to securely connect small, low-power devices to AWS cloud services
- ModusToolbox™ tools and resources can also be used in the command line. See [Running ModusToolbox™ from the Command Line](#) for detailed documentation
- See [AN228571 – Getting Started with PSoC™ 6 MCU on ModusToolbox™](#) for more information
- **PSoC™ Creator:** A proprietary IDE that runs on Windows only. It supports a subset of PSoC™ 6 MCU devices as well as other PSoC™ device families such as PSoC™ 3, PSoC™ 4, and PSoC™ 5LP

5.2.2.1 Choosing an IDE

ModusToolbox™, the latest-generation toolset, includes the ModusToolbox™ IDE. The IDE is Eclipse-based and therefore is supported across Windows, Linux, and MacOS platforms. The tool supports all PSoC™ 6 MCU devices. The associated hardware and middleware configurators also work on all three host operating systems. Certain features of PSoC™ 6 MCU such as UDBs and USB host are not currently supported in ModusToolbox™ IDE. New versions of ModusToolbox™ will be released in the future to support these features and improve the user experience.

Choose ModusToolbox™ if you have prior experience with Eclipse-based tools and want to take advantage of the power and extensibility of an Eclipse-based IDE, or if you want your development environment on Linux or macOS. You should also choose ModusToolbox™ if you want to build an IoT application using IoT devices, or if you are using a PSoC™ 6 MCU device not supported on PSoC™ Creator.

PSoC™ Creator is the long-standing proprietary tool that runs on Windows only. This mature IDE includes a graphical editor that supports schematic based design entry with the help of Components. PSoC™ Creator supports all PSoC™ 3, PSoC™ 4, PSoC™ 5LP devices and a subset of PSoC™ 6 MCU devices. The subset of PSoC™ 6 MCU devices includes devices up to 1 MB of flash.

Choose PSoC™ Creator if you are inclined towards using a graphical editor for design entry and code generation, and if the PSoC™ MCU that you are planning to use is supported by the IDE or if you are intending to use the UDBs on the PSoC™ MCU.

5.2.2.2 PSoC™ Creator

PSoC™ Creator is a free Windows-based Integrated Design Environment (IDE). It brings together several digital, analog, and system Components and firmware to build an application, and enables you to design hardware and firmware systems concurrently. Using PSoC™ Creator, you can select, place, and configure Components on a schematic; write C/Assembly source code; and program and debug the device.

As [Figure 29](#) shows, with PSoC™ Creator, you can:

1. Browse the collection of code examples from the **File > Code Example...** menu
 - a. Filter for examples based on device family
 - b. Select from the menu of examples offered based on the **Filter by** options.
 - c. Download the code example using the download button
 - d. Create a new project based on the selection
2. Explore the library of more than 100 Components
3. Drag and drop Components to build your hardware system design in the main design workspace
4. Review the Component datasheets
5. Configure the Components using configuration tools
6. Co-design your application firmware with the PSoC™ hardware

5 PSoC™ 6 application notes

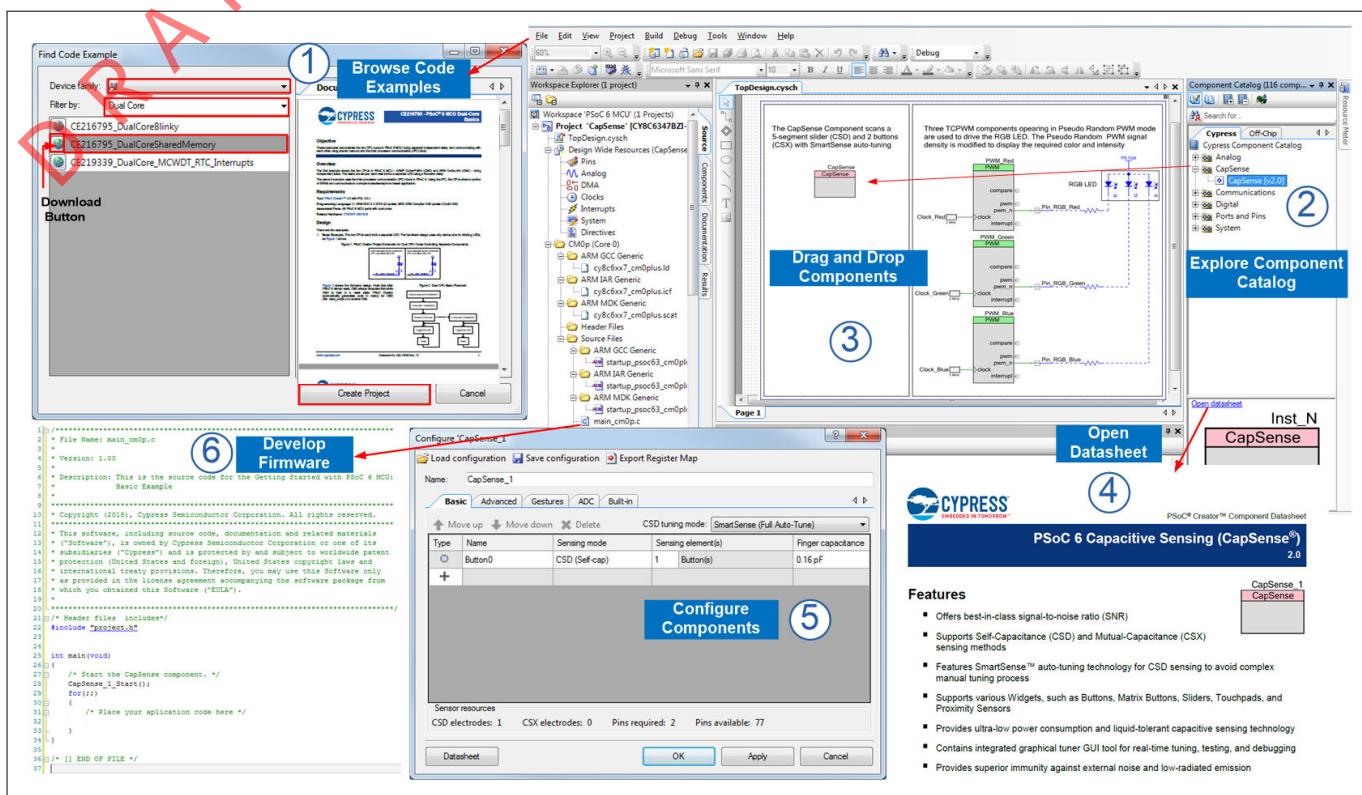


Figure 29 PSoC™ Creator schematic entry and components

PSoC™ Creator help

Visit the [PSoC™ Creator](#) home page to download and install the latest version of PSoC™ Creator. Launch PSoC™ Creator and navigate to the following items:

- Quick Start Guide:** Choose **Help > Documentation > Quick Start Guide**. This guide gives you the basics for developing PSoC™ Creator projects
- Code Examples:** Choose **File > Code Example** or click the **Find Code Example...** link on the **Start Page** tab. These code examples demonstrate how to configure and use PSoC™ resources
- Component Datasheets:** Right-click a Component and select **Open Datasheet**. Visit the [PSoC™ 6 MCU Component Datasheets](#) page for a list of all Component datasheets

5.2.2.2.3 Software development kits for PSoC™ 6 devices

Significant source code and tools are provided to enable software development for PSoC™ 6 MCU. You use tools to specify how you want to configure the hardware, generate code for that purpose which you use in your firmware, and include various middleware libraries for additional functionality, like Bluetooth® Low Energy (LE) connectivity or FreeRTOS. This source code makes it easier to develop the firmware for supported devices. It helps you quickly customize and build firmware without the need to understand the register set.

For the PSoC™ Creator environment, Infineon provides the Peripheral Driver Library (PDL). The PDL supports both PSoC™ Creator and third-party IDEs. You use PSoC™ Creator Components to configure the hardware. PSoC™ Creator generates configuration code based on your choices. That code is based on the source code in the PDL drivers. The PDL also includes various middleware libraries. There may or may not be a Component to assist in configuring that code.

The driver code is delivered as the psoc6pdl library. Middleware is delivered as psoc6mw. The PDL source code is essentially identical, whether delivered with PSoC™ Creator or ModusToolbox IDE. There are implementation differences for the two IDEs.

5 PSoC™ 6 application notes

~~DRAFT~~

There are differences in how the middleware is provided. For example, CapSense functionality is provided in PSoC™ Creator as a Component. For ModusToolbox software, there is a Configurator and a middleware library. See the respective documentation for the two IDEs for details of what's the same, and what's different.

Whether you use PSoC™ Creator or a third-party IDE, firmware developers who wish to work at the register level should also use the driver source code from the PDL. The PDL includes all the device-specific header files and startup code you need for your project. It also serves as a reference for each driver. Because the PDL is provided as source code, you can see how it accesses the hardware at the register level.

Some devices do not support particular peripherals. The PDL is a superset of all the drivers for any supported device. This superset design means:

- All API elements needed to initialize, configure, and use a peripheral are available
- The PDL is useful across various PSoC™ 6 MCU devices, regardless of available peripherals
- The PDL includes error checking to ensure that the targeted peripheral is present on the selected device

This enables the code to maintain compatibility across some members of the PSoC™ 6 device family as long as the peripherals are available. A device header file specifies the peripherals that are available for a device. If you write code that attempts to use an unsupported peripheral, you will get an error at compile time. Before writing code to use a peripheral, consult the datasheet for the particular device to confirm support for the peripheral.

PSoC™ Creator provides Components that are based on the PDL. This retains the essence of PSoC™ Creator in utilizing Infineon or community-developed and pre-validated Components. However, the PDL is a source code library that you can use with any development environment.

The PDL includes the following key software resources:

- Header and source files for each peripheral driver
- Header and source files for middleware libraries
- Device-specific header, startup, and configuration files
- Template projects for supported third-party IDEs
- Full documentation, available in <PDL install directory>\doc\

There are two key documents:

The PDL v3.x User Guide covers the fundamentals of working with the PDL, such as the following:

- Creating a custom project using the PDL (including third-party IDEs)
- Configuring a peripheral
- Managing pins in firmware
- Using the PDL as a learning tool for register-based programming
- Using the PDL API Reference documentation

The PDL 3.x API Reference Manual.html. This reference has complete information on every driver in the PDL, including overview, configuration considerations, and details on every function, macro, data structure, and enumerated type.

5.2.2.3 Support for other IDEs

You can also develop firmware for PSoC™ 6 MCU using your favorite IDE such as [IAR Embedded Workbench](#).

It is recommended that you generate resource configuration using a configuration tool. For PSoC™ Creator, configuration is integral to the IDE.

PSoC™ Creator is used to set up and configure PSoC™ 6 MCU system resources and peripherals. You then export the project to your IDE, and continue developing firmware in your IDE. If there is a change in the device configuration, you edit the TopDesign schematic in PSoC™ Creator and regenerate the code for the target IDE.

You can work effectively in most if not all IDEs. If your IDE is not supported in the Target IDEs panel, you can still use PSoC™ Creator. After you generate code, add the necessary files directly to your IDE's project. [AN219434](#)

5 PSoC™ 6 application notes

~~DRAFT~~
- [PSoC™ 6 MCU Importing Generated Code into an IDE](#) provides detailed steps for manually importing the generated code into another IDE.

5.2.2.4 RTOS support

5.2.2.4.1 RTOS support with PSoC™ Creator

The PDL includes RTOS support for PSoC™ 6 MCU development: FreeRTOS source code is fully integrated and included with the PDL. You can import the FreeRTOS software package into your project by using the PSoC™ Creator RTOS import option. Navigate to the **Project > Build Settings** menu and select **FreeRTOS** from the **Software package imports** option under **Peripheral Driver Library > FreeRTOS** as shown in [Figure 30](#).

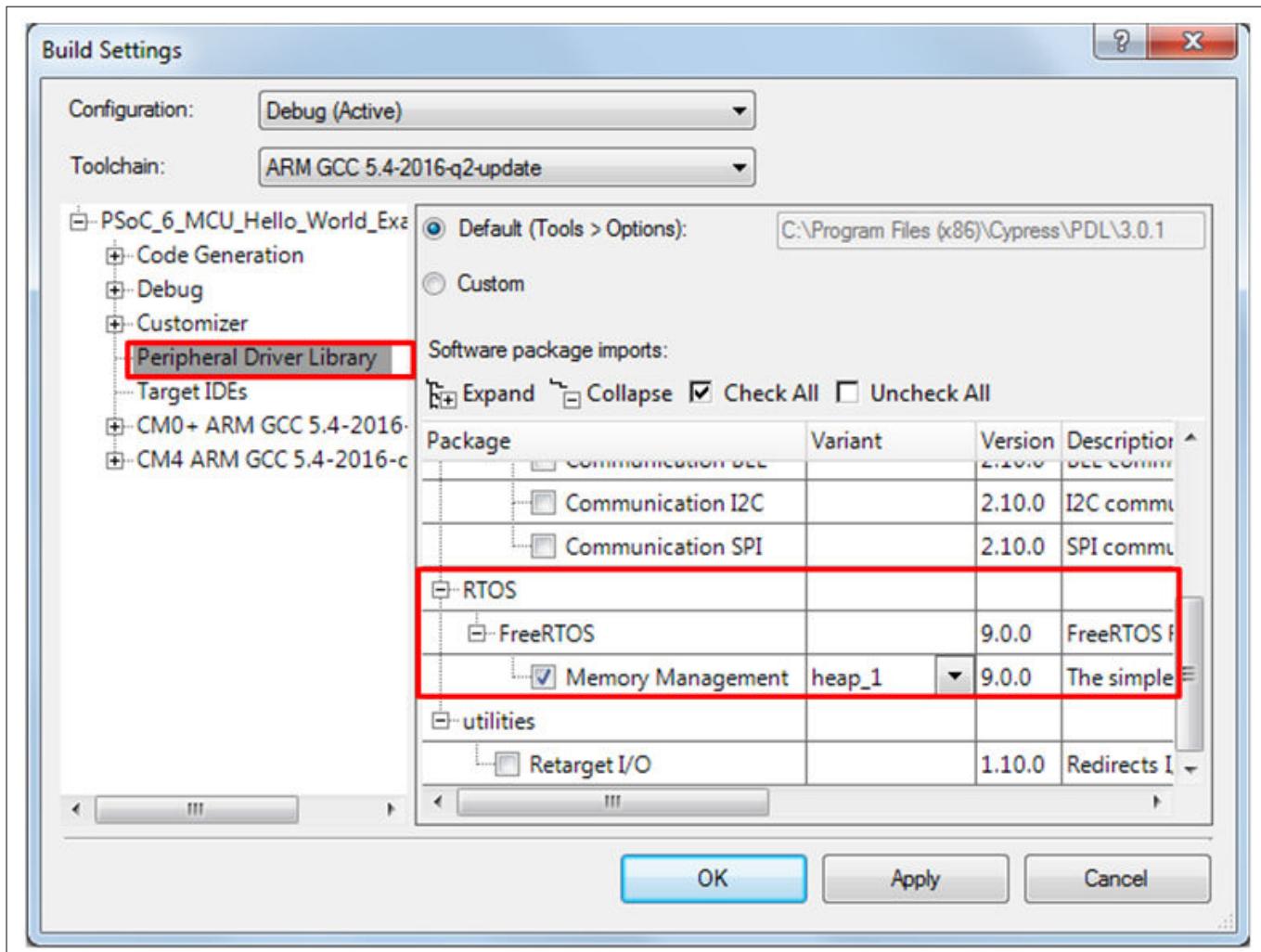


Figure 30 Import FreeRTOS in PSoC™ creator project

If you have a preferred RTOS, use the resources provided as examples on how to integrate such code with the PDL.

5.2.2.5 Programming/debugging

The [PSoC™ 6 Wi-Fi-BT Pioneer Kit \(CY8CKIT-062-WiFi-BT\)](#) and [PSoC™ 6 BLE Pioneer Kit \(CY8CKIT-062-BLE\)](#) have the KitProg2 onboard programmer/debugger. It supports Cortex Microcontroller Software Interface Standard - Debug Access Port (CMSIS-DAP) and custom modes of operations, as well as the KitProg2 connection. This makes debugging the PSoC™ 6 MCU Pioneer Kit extremely flexible. See the [KitProg2 User Guide](#) for details.

5 PSoC™ 6 application notes

The [PSoC™ 6 Wi-Fi BT Prototyping Kit \(CY8CPROTO-062-4343W\)](#) has the KitProg3 onboard programmer/debugger. It supports Cortex Microcontroller Software Interface Standard - Debug Access Port (CMSIS-DAP). See the [KitProg3 User Guide](#) for details.

PSoC™ Creator supports debugging a single CPU (either Cortex-M4 or Cortex-M0+) at a time. Some third-party IDEs support multi-CPU debugging. For more information on debugging firmware on PSoC™ devices with PSoC™ Creator, refer to the PSoC™ Creator Help.

5.2.2.6 PSoC™ 6 MCU development kits

PSoC™ 6 Wi-Fi-BT Pioneer Kit ([CY8CKIT-062-WiFi-BT](#)) and PSoC™ 6 Wi-Fi BT Prototyping Kit ([CY8CPROTO-062-4343W](#)) are development kits that support the PSoC™ 62 series MCU along with Wi-Fi and Bluetooth® connectivity.

The [PSoC™ 6 BLE Pioneer Kit \(CY8CKIT-062-BLE\)](#) and [PSoC™ 6 BLE Prototyping Kit \(CY8CPROTO-063-BLE\)](#) support the PSoC™ 6 MCU with Bluetooth® LE Connectivity. Refer to the [PSoC™ 6 MCU product page](#) for more information.

~~5 PSoC™ 6 application notes~~~~5.2.3 Device features~~

The PSoC™ 6 MCU device has an extensive feature set as shown in [Figure 31](#). The following is a list of its major features. For more information, see the device [datasheet](#), the [Technical Reference Manual \(TRM\)](#), and the section on [Related application notes and code examples](#).

- MCU subsystem
 - 150-MHz Arm® Cortex®-M4 and 100-MHz Arm® Cortex®-M0+
 - Up to 2 MB of flash with additional 32 KB for EEPROM emulation and 32KB supervisory flash
 - Up to 1 MB of SRAM with selectable Deep Sleep retention granularity at 32-KB retention boundaries
 - Inter-processor communication supported in hardware
 - DMA controllers
- Security Features
 - Cryptography accelerators and true random number generator function
 - One-time programmable eFUSE for secure key storage
 - Secure boot with hardware hash-based authentication
- I/O subsystem
 - Up to 104 GPIOs with programmable drive modes, drive strength, slew rates
 - Two ports with Smart I/O that can implement Boolean operations
- Programmable analog blocks
 - Two opamps of 6-MHz gain bandwidth (GBW) and two low-power comparators
 - Up to One 12-bit, 1-MspS SAR ADC and one 12-bit voltage-mode DAC
- Programmable digital blocks, communication interfaces
 - Up to 12 UDBs for custom digital peripherals
 - Up to 32 TCPWM blocks configurable as 16-bit/32-bit timer, counter, PWM, or quadrature decoder
 - Up to 13 SCBs configurable as I2C Master or Slave, SPI Master or Slave, or UART
 - Controller Area Network interface with Flexible Data-Rate
 - Up to two Secure Digital Host Controllers with support for SD, SDIO, and eMMC interfaces
 - Audio subsystem with up to two I2S interface and two PDM channels
 - SMIF interface with support for execute-in-place from external quad SPI flash memory and on-the-fly encryption and decryption
 - Full-Speed, dual-role USB with device and host capability
- CAPSENSE™ with SmartSense auto-tuning
 - Supports both CAPSENSE™ Sigma-Delta (CSD) and CAPSENSE™ Transmit/Receive (CSX) controllers
 - Provides best-in-class SNR, liquid tolerance, and proximity sensing
- Operating voltage range, power domains, and low-power modes
 - Device operating voltage: 1.71 V to 3.6 V with user-selectable core logic operation at either 1.1 V or 0.9 V
 - Multiple on-chip regulators: low-drop out (LDO for Active, Deep Sleep modes), buck converter
 - Six power modes for fine-grained power management
 - An “Always ON” backup power domain with built-in RTC, power management integrated circuit (PMIC) control, and limited SRAM backup

5 PSoC™ 6 application notes

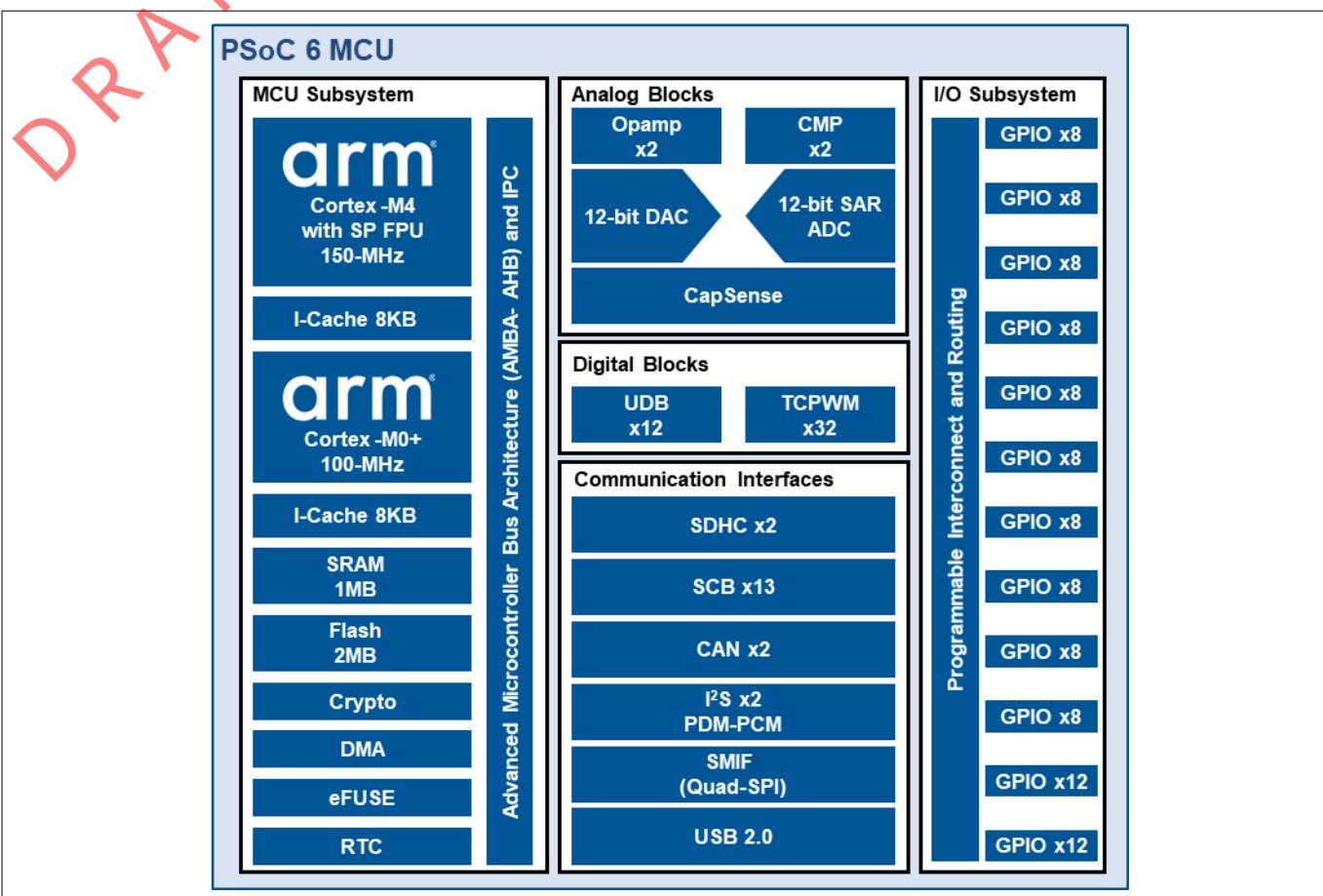


Figure 31 PSoC™ 6 MCU block diagram

5 PSoC™ 6 application notes

5.2.4 My first PSoC™ 6 MCU design using PSoC™ Creator

This section does the following:

- Demonstrates how to build a simple PSoC™ 6 MCU-based design and program it on to the development kit
- Provides detailed steps that make it easy to learn PSoC™ 6 MCU design techniques and how to use the PSoC™ Creator IDE

5.2.4.1 Using these instructions

These instructions are grouped into several sections. Each section is devoted to a particular phase of the application development workflow. The major sections are:

- Part 1: Create a new project from scratch
- Part 2: Implement the design
- Part 3: Generate source code
- Part 4: Write the firmware
- Part 5: Build the project and program the device
- Part 6: Test your design

If you are familiar with developing projects with PSoC™ Creator, you can use the PSoC™ Creator version of the code example [CE221773 – PSoC™ 6 MCU Hello World Example](#) directly. It is a complete design, with all the firmware written. You can walk through the instructions and observe how the steps are implemented in the code example.

If you start from scratch and follow all the instructions in this application note, you use the code example as a reference while following the instructions.

You can download the code example from the website by clicking the link above. You can also use the PSoC™ Creator **File > Code Example** command. Set the **Device family** to PSoC™ 62. Select the PSoC™ MCU Hello World Example. Download the code example by clicking on the download icon adjacent to the example and then click on **Create Project**, and follow the on-screen instructions.

This design is developed for the [CY8CKIT-062-WiFi-BT PSoC™ 6 Wi-Fi-BT Pioneer Kit](#). You can also use [CY8CKIT-062-BLE PSoC™ 6 BLE Pioneer Kit](#) to test this example by selecting the appropriate device from the Device Selector.

5.2.4.2 About the design

This design uses the CM4 CPU of PSoC™ 6 MCU to execute two tasks: UART communication and LED control. At device reset, the CM0+ CPU enables the CM4 CPU. The CM4 CPU uses the UART Component to print a “Hello World” message to the serial port stream and when the Enter Key is pressed by the user, the LED on the PSoC™ 6 MCU Wi-Fi-Bluetooth® Pioneer Kit starts blinking.

5 PSoC™ 6 application notes

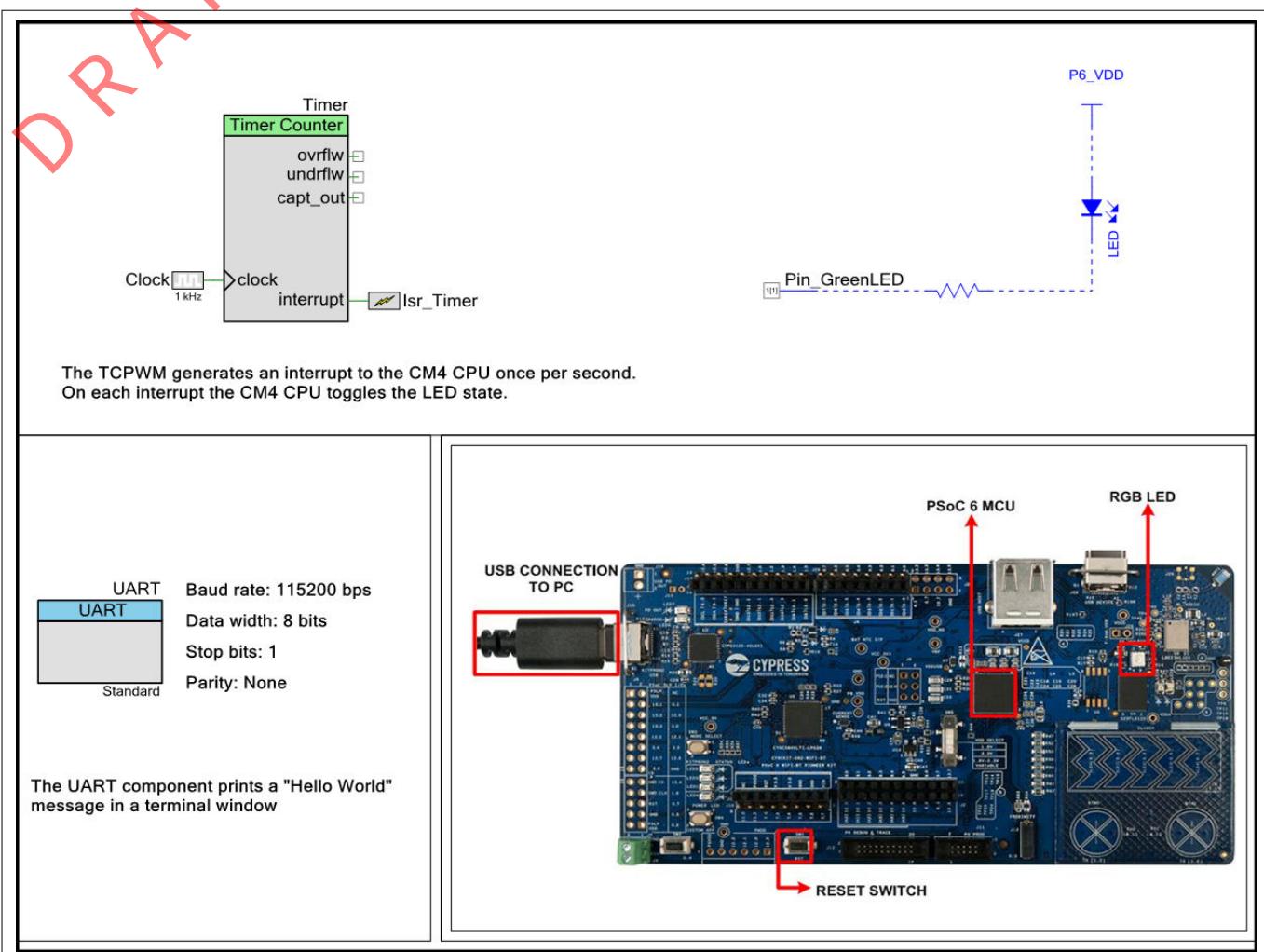


Figure 32 My first PSoC™ 6 MCU design

5.2.4.3 Part 1: Create a new project from scratch

This section takes you on a step-by-step guided tour of the design process. It starts with creating an empty project and guides you through hardware and firmware design development stages.

Note: These instructions assume that you are using PSoC™ Creator 4.2. The overall development process is the same for subsequent versions of PSoC™ Creator; but the user interface may change over time.

Launch PSoC™ Creator and get started.

1. Ensure that PSoC™ Creator can find the PDL

This should be set correctly automatically during installation, but nothing works if this isn't set up right. Refer to [Figure 33](#) for help with this step.

- Choose **Tools > Options**
- On the **Project Management** panel, check the path in the **PDL v3 (PSoC™ 6 Devices) location** field
- Ensure that it is correct. If it is not, click the **Browse** button and locate the installed directory of the PDL. The default location is C:\Program Files (x86)\Cypress\PDL\3.0.1

5 PSoC™ 6 application notes

DRAFT

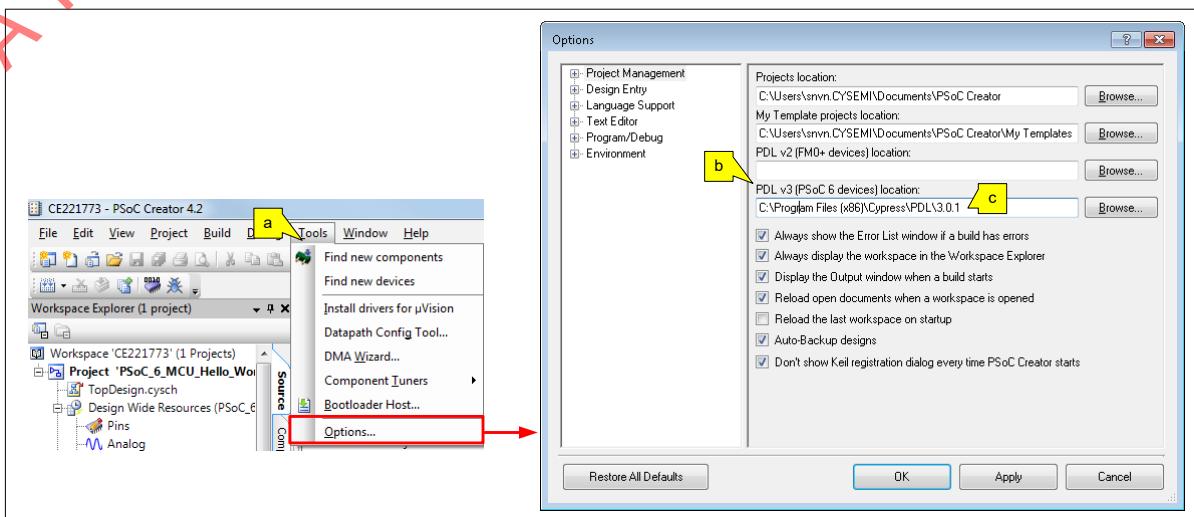


Figure 33 Peripheral driver library (PDL) location

Optional: Jump to [Part 2: Implement the design](#).

2. Create a new PSoC™ Creator project

Choose **File > New > Project**, as [Figure 34](#) shows. The **Create Project** window appears.

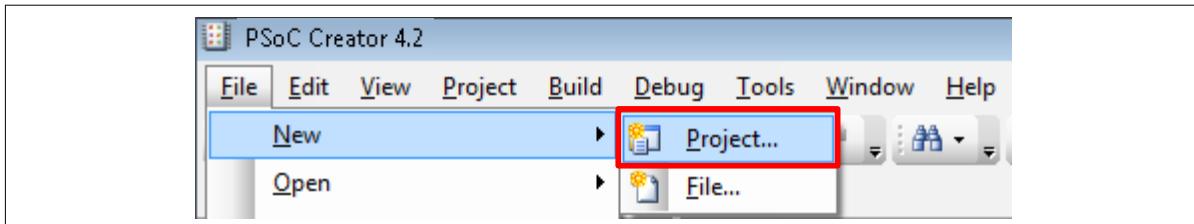


Figure 34 Create a New PSoC™ Creator Project

Note: If you are using the code example, choose **File > Open > Project/Workspace**, as [Figure 35](#) shows. The **Open** window appears. Point to the location of the code example workspace and open the workspace.

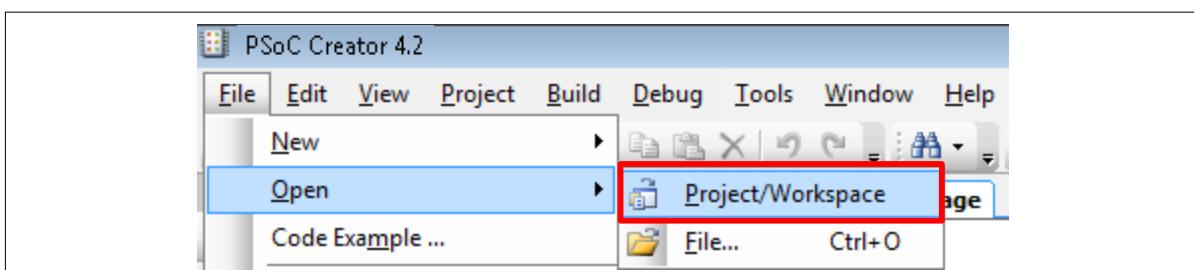


Figure 35 Open existing code example workspace

3. Select PSoC™ 6 MCU as the target device

PSoC™ Creator speeds up the development process by automatically setting various project options for specified development kits or target devices. See [Figure 36](#) for help with this step.

- Click **Target device**
- In the family drop-down menu, select **PSoC 6**
- In the device drop-down menu, select **PSoC 62**
- Click **Next**. The Select project template panel appears

5 PSoC™ 6 application notes

~~DRAFT~~
PSoC™ Creator uses CY8C6247BZI-D54 as the default device in the PSoC™ 6 MCU family. This device is mounted on the [CY8CKIT-062-WiFi-BT PSoC™ 6 Wi-Fi-BT Pioneer Kit](#).

If you are using custom hardware based on PSoC™ 6 MCU, or a different PSoC™ 6 MCU part number, this is the place you choose to Launch Device Selector option in Target device and select the appropriate part number.

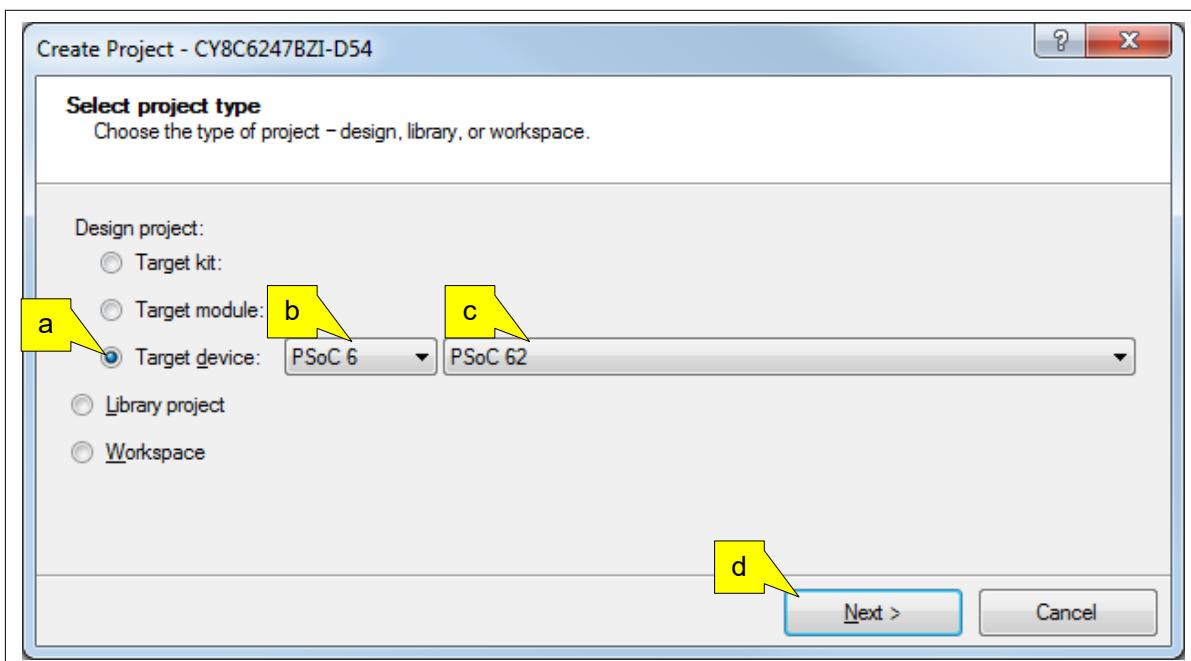


Figure 36 Selecting target device

4. Pick a project template

- a. Choose **Empty Schematic**
- b. Click **Next**

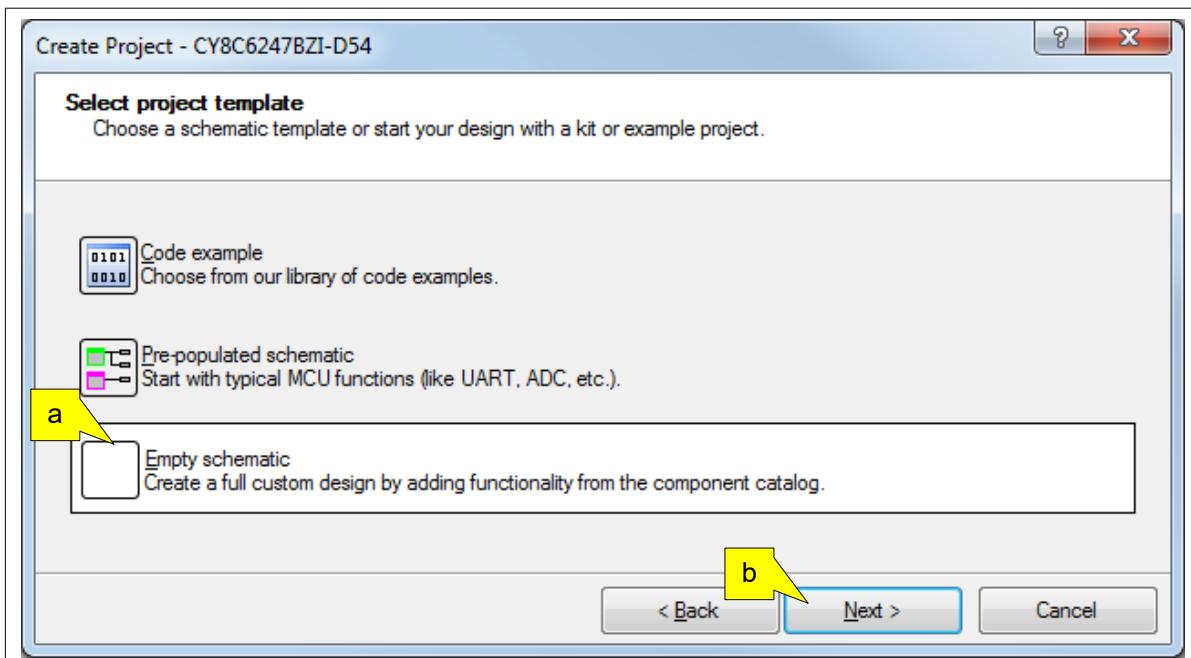


Figure 37 Pick a project template

5. Select target IDE(s)

5 PSoC™ 6 application notes

If you expect to export the code from the project, specify the target IDE. By default, all export options are disabled. You can modify this setting later if circumstances change.

Click **Next** to accept the default options.

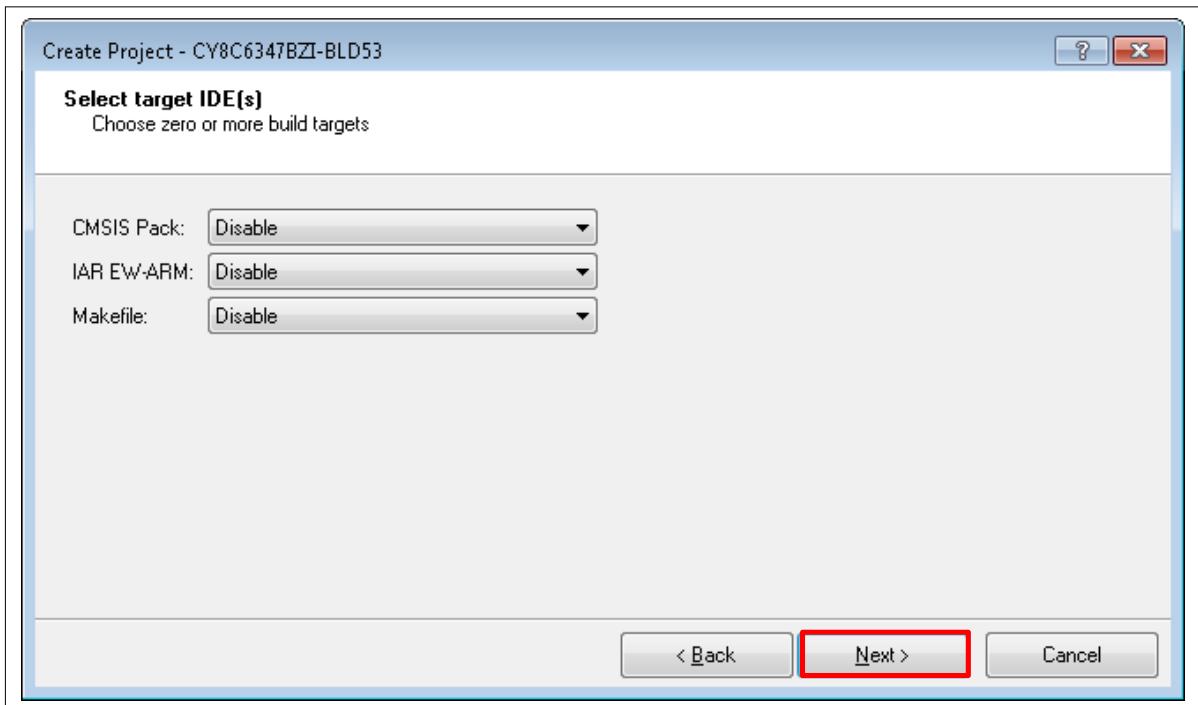


Figure 38 Select target IDEs (All disabled)

6. Create the project

In this step, you set the name and location for your workplace, and a name for the project. See [Figure 39](#) for help with this step. A workspace is a container for one or more projects.

- a. Set the Workspace name
- b. Specify the **Location** of your workspace
- c. Set a **Project name**. The project and workspace names can be the same or different
- d. Click **Finish**

5 PSoC™ 6 application notes

DRAFT

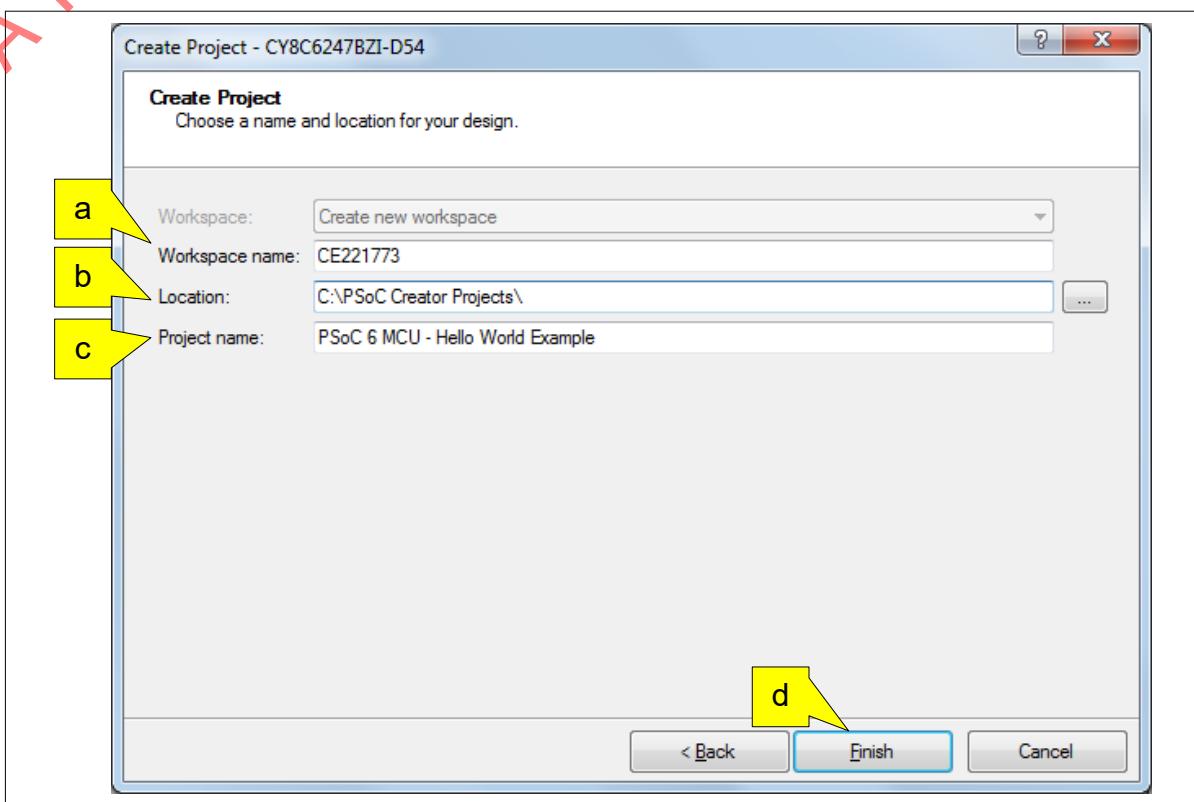


Figure 39 Project naming and location

You have successfully created a new PSoC™ Creator project.

5.2.4.4 Part 2: Implement the design

Now that you have a project file, it is time to implement the hardware design using PSoC™ Creator Components. If you are using the code example directly, you already have a complete design.

Before you implement the design, a quick tour of the PSoC™ Creator interface is in order.

Figure 40 shows the PSoC™ Creator application displaying an empty design schematic.

The project includes a project folder with a base set of files. You view these files in the **Workspace Explorer** pane to the left. The project schematic opens by default. This is the TopDesign.cysch file. Double-click the file name in the explorer pane to open the schematic at any time. In a new project, the schematic is empty. If you are using the code example, this is the schematic for the design.

The Component catalog is on the right side of the window. You can open it with the **View > Component Catalog** menu item. You can search for a particular Component by typing the name of the Component in the **Search for...** text box and then pressing the enter key. See Figure 40.

5 PSoC™ 6 application notes

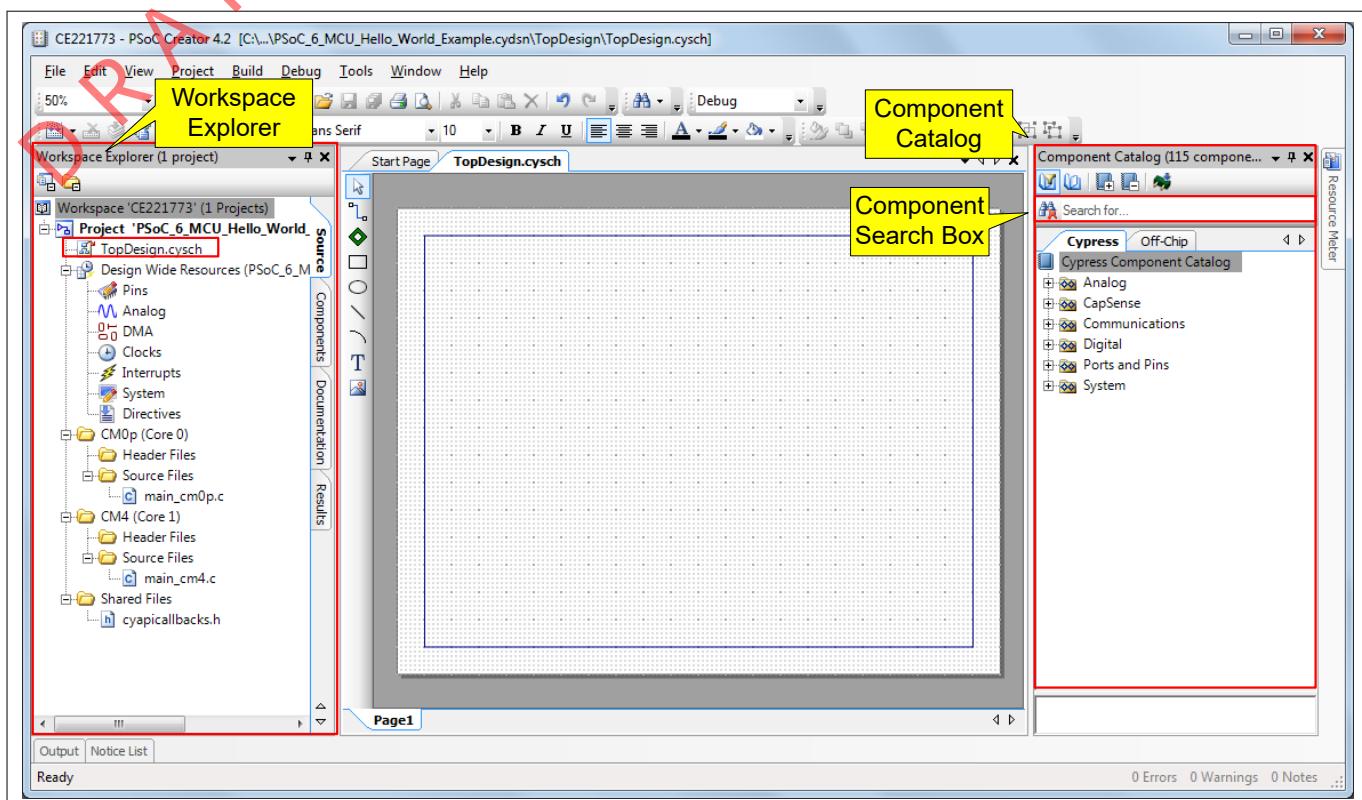


Figure 40 Schematic and component catalog

1. Place Components in the design

This design uses several Components: three digital output pins, a UART, a Watchdog Timer, and an Interrupt. In this step, you add them to the design. You configure them in subsequent steps. [Figure 41](#) shows the result

- In the **Component Catalog**, expand the **Communications** group, drag a **UART (SCB)** Component into the schematic, and drop it. It doesn't matter where you put a Component
- Expand the **Ports and Pins** group, and drag a **Digital Output Pin** into the design
- Expand the **Digital** group, and drag a **Timer Counter (TCPWM)** Component into the design
- Expand the **System** group, and drag an **Interrupt** Component and a **Clock** Component into the design

5 PSoC™ 6 application notes

DRAFT

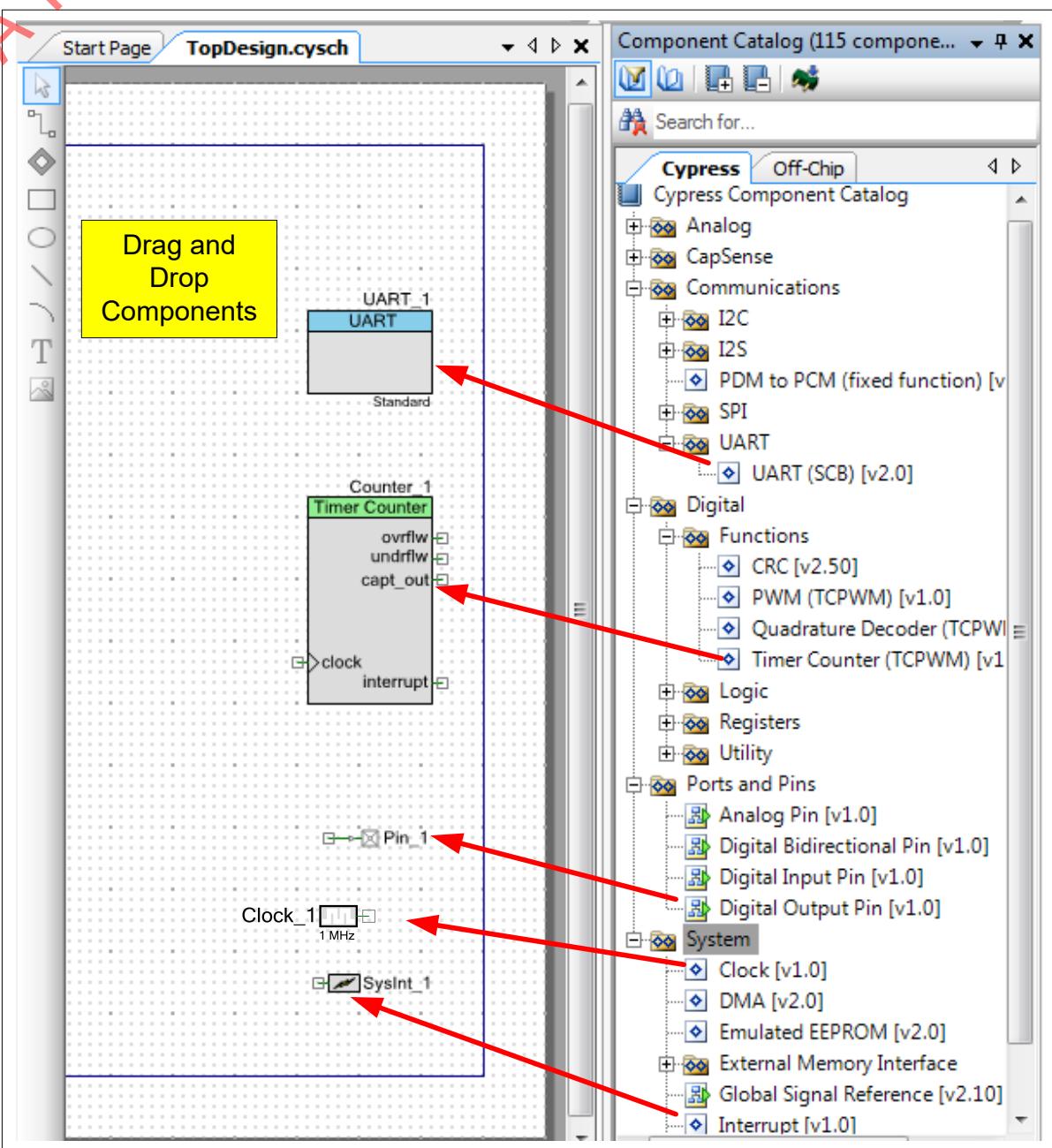


Figure 41 Place components in the design

PSoC™ Creator gives each Component a default name and properties. Default values may or may not be suitable for any given design. In subsequent steps, you modify the name and some of the properties

2. Configure the LED pin

The output pin drives the LED. The LED on the PSoC™ 6 Wi-Fi-BT Pioneer Kit is active LOW; that is, the logic HIGH pin-drive state turns OFF the LED, and the logic LOW pin-drive state turns it ON. [Figure 42](#) shows the configuration

Double-click the Component placed on the schematic to open the configuration dialog. Then, perform the following steps:

- Change the name of the Component instance to **Pin_GreenLED**
- Deselect **HW connection**. The firmware will drive the pin
- Set the **Drive Mode** to Resistive Pull Up

5 PSoC™ 6 application notes

DRAFT

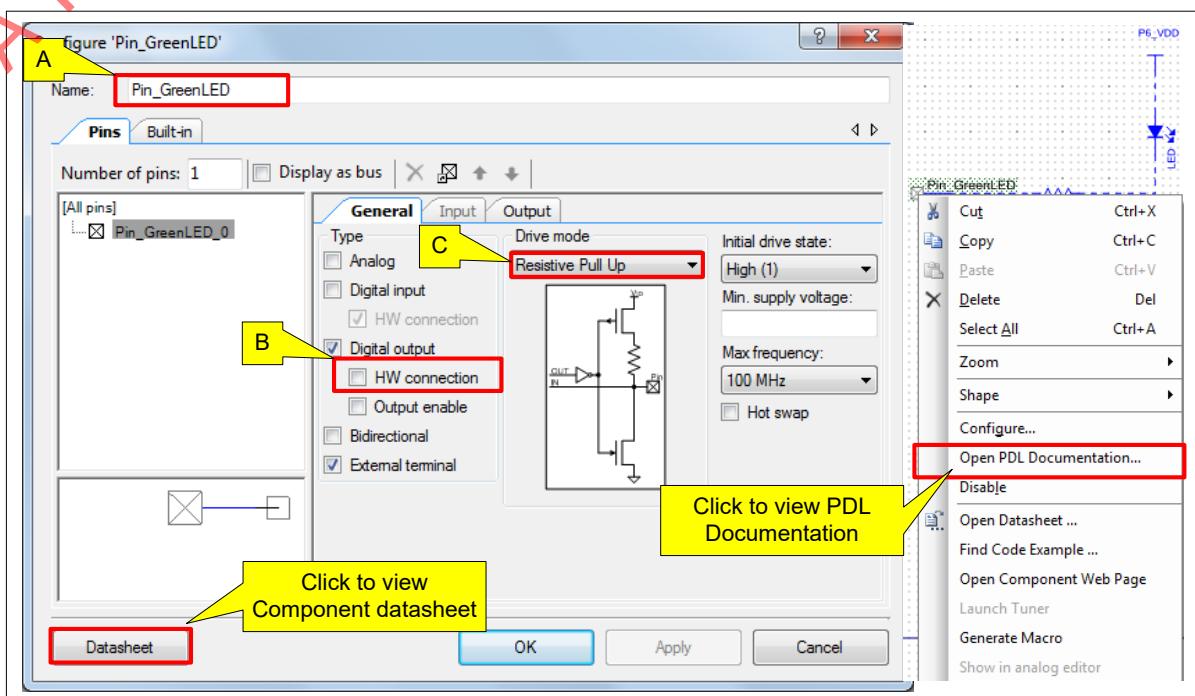


Figure 42 Configuring an output pin component

Notes:

- Each Component has an associated datasheet that can be accessed from the configuration window. The Component datasheet provides more information on the Component configuration, the application programming interface (API), and the electrical specifications
- You can open the API reference document of the associated PDL driver of a Component by right-clicking the Component and clicking on **Open PDL Documentation...** link. See [Figure 42](#).
- For a pin, if you enable **External terminal**, you can add external “off-chip” Components to a design. External Components on the schematic are included for descriptive purposes only; they have no effect on the generated code. Off-chip Components are optional, but can assist the hardware design team understanding how the design works. You can also add text boxes to a design with descriptions. [Figure 43](#) shows how you could enhance the design for the LED. In this case, the off-chip components were configured with the **Instance_Name_Visible** option unchecked. The resistor was configured with the **Value** field left blank. The power terminal was configured with the **Supply_Name** set to **P6_VDD**

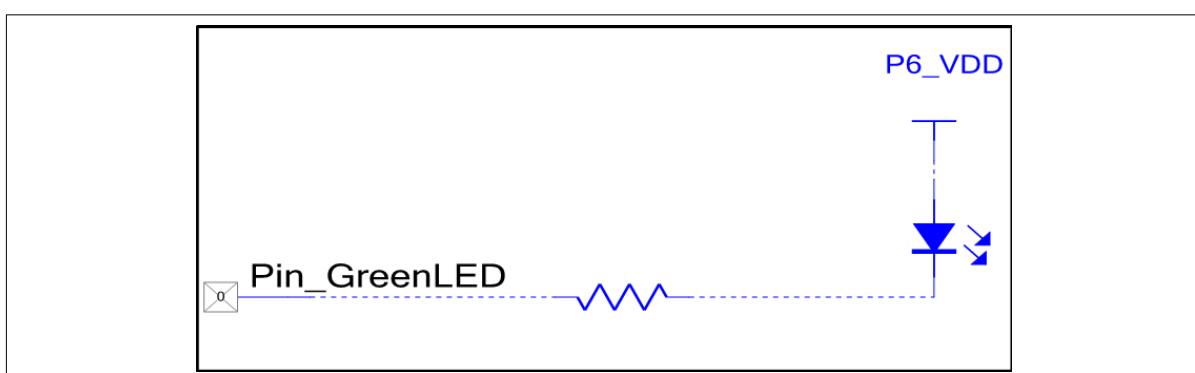


Figure 43 An output pin with off-chip components

3. Configure the UART Component

5 PSoC™ 6 application notes

~~DRAFT~~
Double-click the Component to open the configuration window. The design uses this Component to display messages in a terminal window at a baud rate of 115200 bps

- Change the **Name** of the Component instance to **UART**.
- Click **OK**.

The design uses default values for all other settings

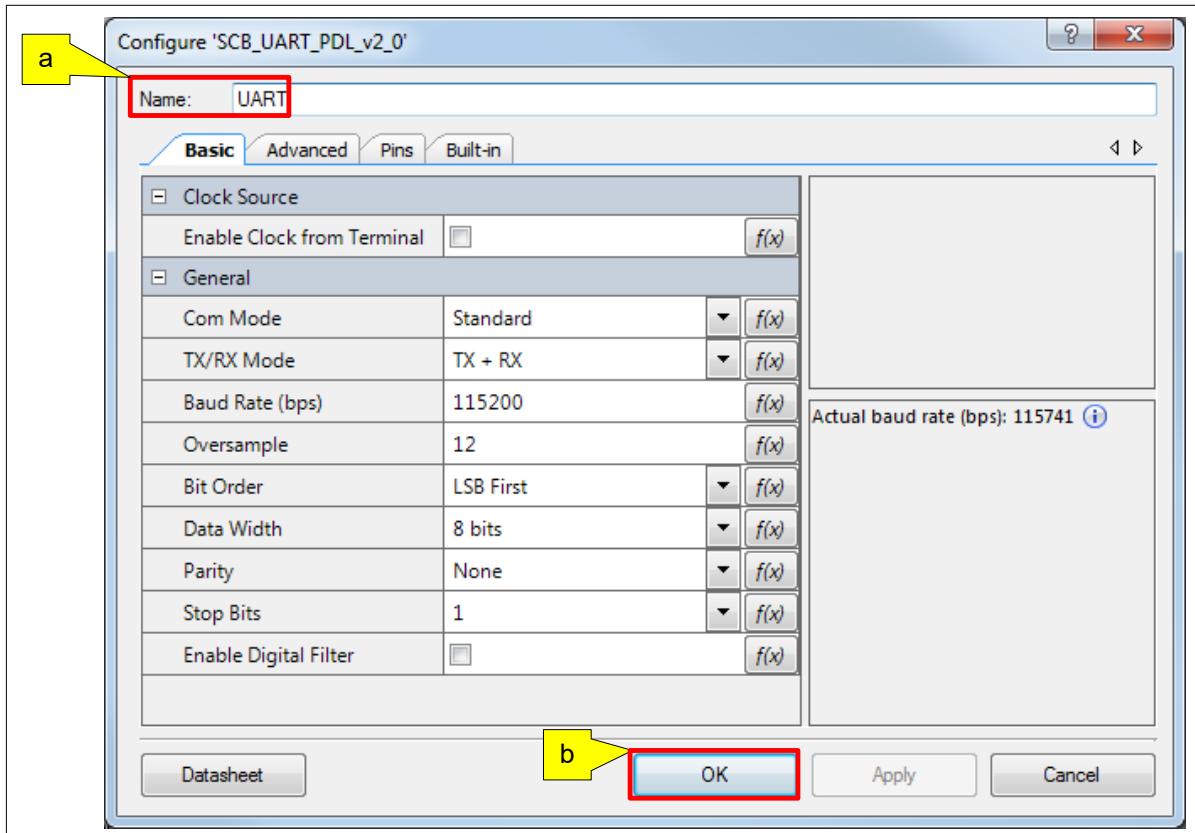


Figure 44 Configuring the SCB-based UART component

4. Configure the Timer Counter (TCPWM) Component to trigger an interrupt

In this step, you configure the Timer Counter (TCPWM) Component to trigger an interrupt every second (1 Hz). The clock source of the TCPWM is the peripheral clock (Clk_Per). The design will use this interrupt to toggle the LED state. Open the Component customizer and follow the steps illustrated in [Figure 45](#).

- Change the **Name** to **Timer**
- Set the **Period** to 1000 and **Interrupt Source** as Overflow/Underflow
- Click **OK** to complete the configuration of the TCPWM Component
- Connect the clock terminal of the TCPWM to the 1-kHz clock source. In the schematic, use the wire tool button or press the 'W' key to start wiring the Clock Component to the clock terminal of the TCPWM Component

5 PSoC™ 6 application notes

DRAFT

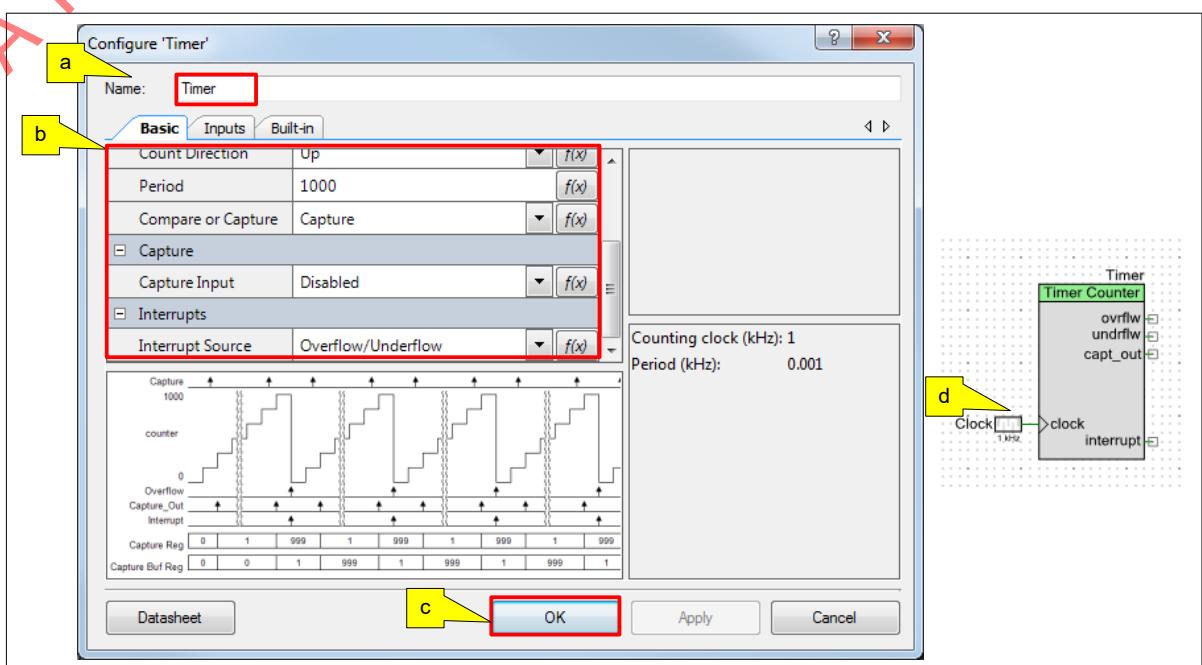


Figure 45 Configuring the TCPWM component

5. Configure the interrupt Component

In this step, you configure the SysInt Component to map the TCPWM interrupt to the CM4 CPU. Open the Component customizer and follow the steps illustrated in [Figure 46](#).

- Change the **Name** to **Isr_Timer**
- Click **OK** to complete the configuration of the SysInt Component

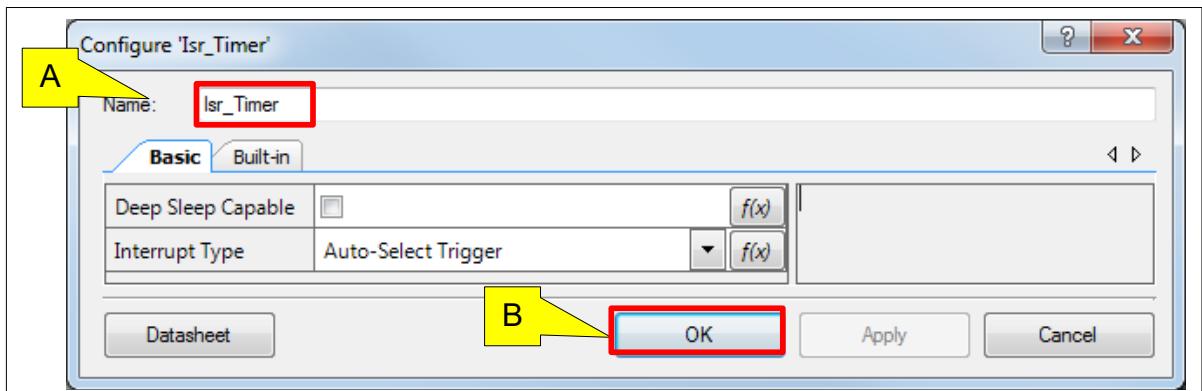


Figure 46 SysInt_PDL Settings

As the final step, connect the interrupt output of the TCPWM Component to the Isr_TCPWM Component input. This routes the TCPWM interrupt to the CM4 CPU (the selection of the CM4 CPU for this interrupt will be set in the system interrupt configuration in a later step). In the schematic, use the wire tool button or press the 'W' key to start wiring the Components

5 PSoC™ 6 application notes

DRAFT

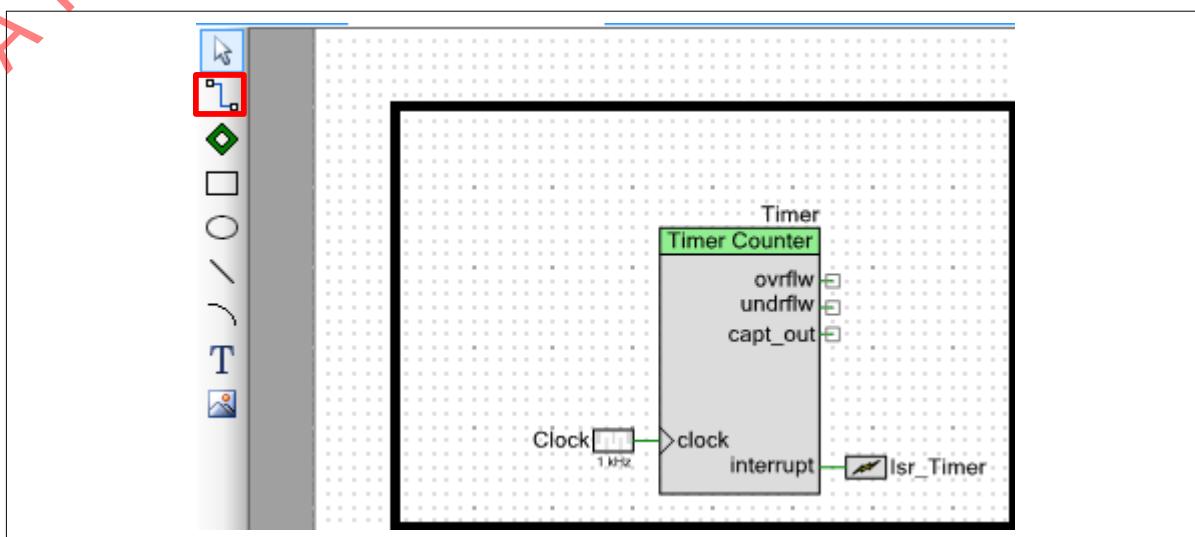


Figure 47 Connect TCPWM peripheral interrupt to CM4 CPU

6. Set the physical pins for each Pin Component

One task remains to complete the design. You must associate each Component with the required physical pins on the device. The choice of which pin to use is driven by the board design. You can find this information in the kit schematic. [Figure 48](#) shows the result of this step. You can connect external LEDs to the selected pins

To set a pin, type either the port number or pin number in the corresponding field, or use the drop-down menu to pick the port or pin. Typically, the port number is used instead of the pin number since these names are independent of the specific package being used.

- Open the pin selector

In the **Workspace Explorer** pane, double-click the **Pins** item under the Design Wide Resources. The pin selector for this device appears

- Set each pin as shown in [Table 5](#)

Table 5 Physical pin assignments for CY8CKIT-062-WiFi-BT pioneer kit

Pin Component Name	Port Name
UART: rx	P5[0]
UART: tx	P5[1]
Pin_GreenLED	P1[1]

5 PSoC™ 6 application notes

DRAFT

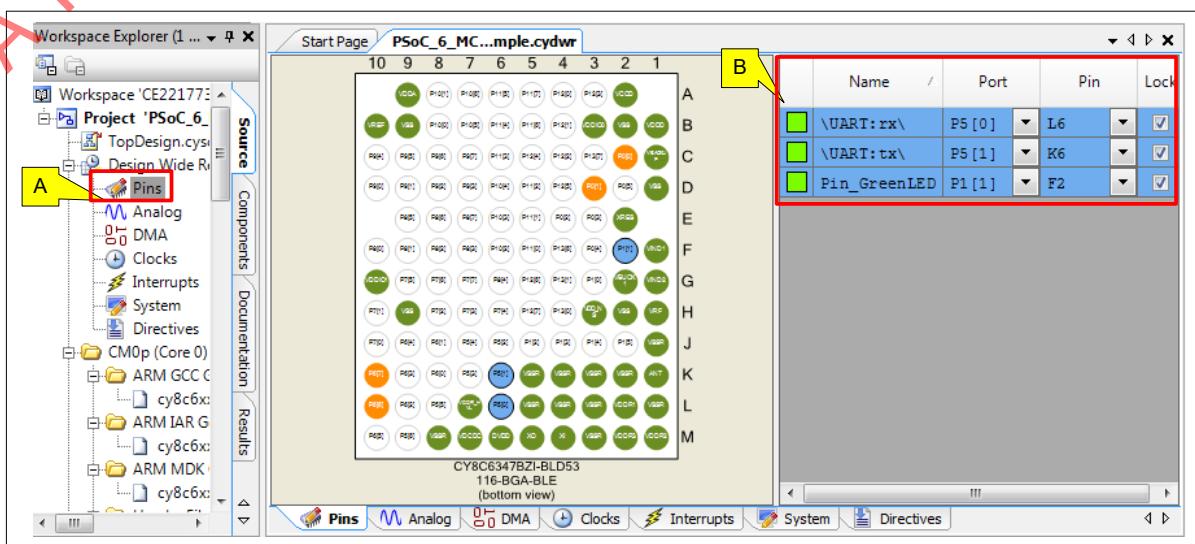


Figure 48 Pin assignment

7. Configure System Clock

The design uses default values for the high-frequency system clock settings. Although you do not modify high frequency clocks for this design, you should know how PSoC™ Creator manages them. If you are working with your own board, you may need to modify these clocks.

- a. In the **Workspace Explorer** pane, double-click the **Clocks** that is under **Design Wide Resources**. The list of clocks appears
- b. Click **Edit Clock**. The **Configure System Clocks** dialog appears
Here, you can see the clock tree, and modify the clocks as required. Note that there are tabs for different types of clocks such as **Source Clocks**, **FLL/PLL**, **High Frequency Clocks**, and **Miscellaneous Clocks**.
- c. Click on the **FLL/PLL** tab. By default, PSoC™ Creator enables FLL and sets the frequency to 100 MHz
- d. Click on the **High Frequency Clocks** tab
- e. You can set the CM4 CPU clock by setting the divider in **Clk_Fast**. By default, the divider is set to 1
- f. You can set the CM0+ CPU clock by setting the divider in **Clk_Slow**. By default, the divider is set to 1. See [Figure 49](#)

5 PSoC™ 6 application notes

DRAFT

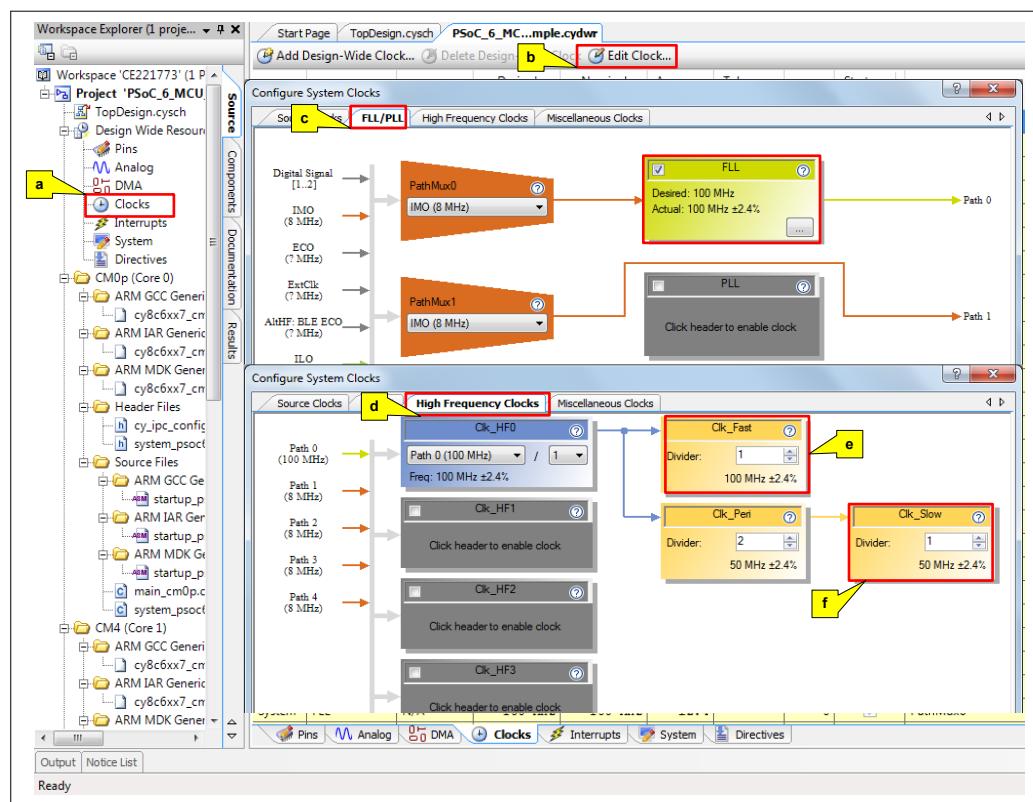


Figure 49 **Clock configuration**

8. Configure System Interrupts

In this step, you configure the system interrupts. See [Figure 50](#).

- In the **Workspace Explorer** pane, double-click the **Interrupts** that is under **Design Wide Resources**. The list of interrupts appears
- Enable **Isr_Timer** for the CM4 CPU

The interrupt numbers are generated automatically by PSoC™ Creator when you generate the code in [Part 3: Generate source code](#).

5 PSoC™ 6 application notes

DRAFT

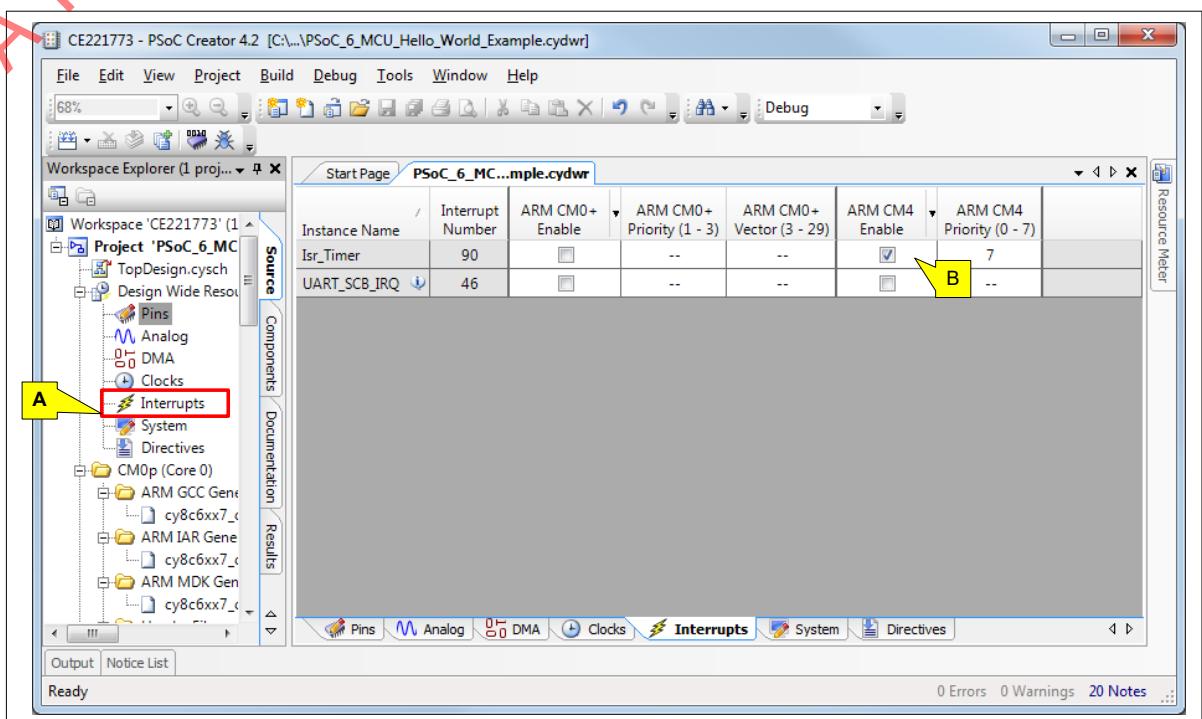


Figure 50 Interrupt configuration

The next part in the development process is to generate code.

Note: This exercise does not detail how to export your work to a target IDE. However, if you wish to use a target IDE, this is the point in the workflow where you would ensure that the correct target IDE is selected before you generate the source code.

5.2.4.5 Part 3: Generate source code

PSoC™ Creator generates the source code based upon the design. The recommended workflow is to generate code before writing firmware. PSoC™ Creator will automatically create macros, constants, and API calls that you may then use in your firmware.

1. Generate the application

Choose **Build > Generate Application**. PSoC™ Creator generates the source code based on the design and puts the files in the Generated_Source folder. See [Figure 51](#). PSoC™ Creator will alert you to errors or problems that may occur. If you are working from scratch and encounter errors, revisit the configuration steps in [Part 2: Implement the design](#) to ensure you have performed them correctly

5 PSoC™ 6 application notes

DRAFT

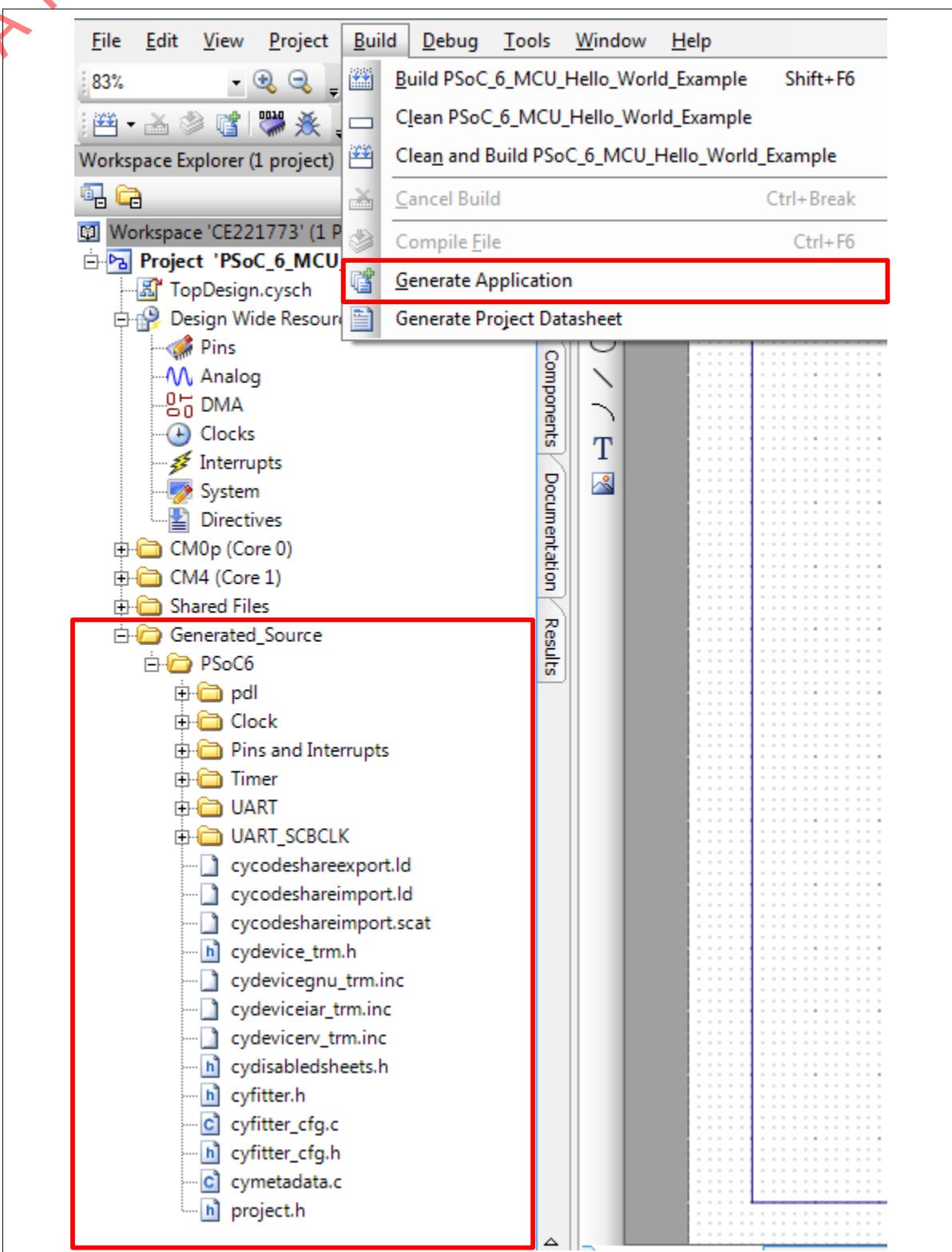


Figure 51 Generate application

Background: PSoC™ 6 MCU is a dual-CPU platform. You can target firmware to run either on Cortex®-M4 or Cortex®-M0+. You set this at the source file level by accessing the file properties. Right-click on a source file, and select **Properties**. Figure 52 shows the **Properties** dialog window. By default, the `main_cm0p.c` file is targeted to the Cortex®-M0+ and the `main_cm4.c` file is targeted to the Cortex®-M4. You do not need to modify the properties for any other file. They are already set in the code example.

5 PSoC™ 6 application notes

By convention, files targeted to run on the CM0+ CPU are located in the cm0p folder and files targeted to run on the CM4 CPU are located in the cm4 folder.

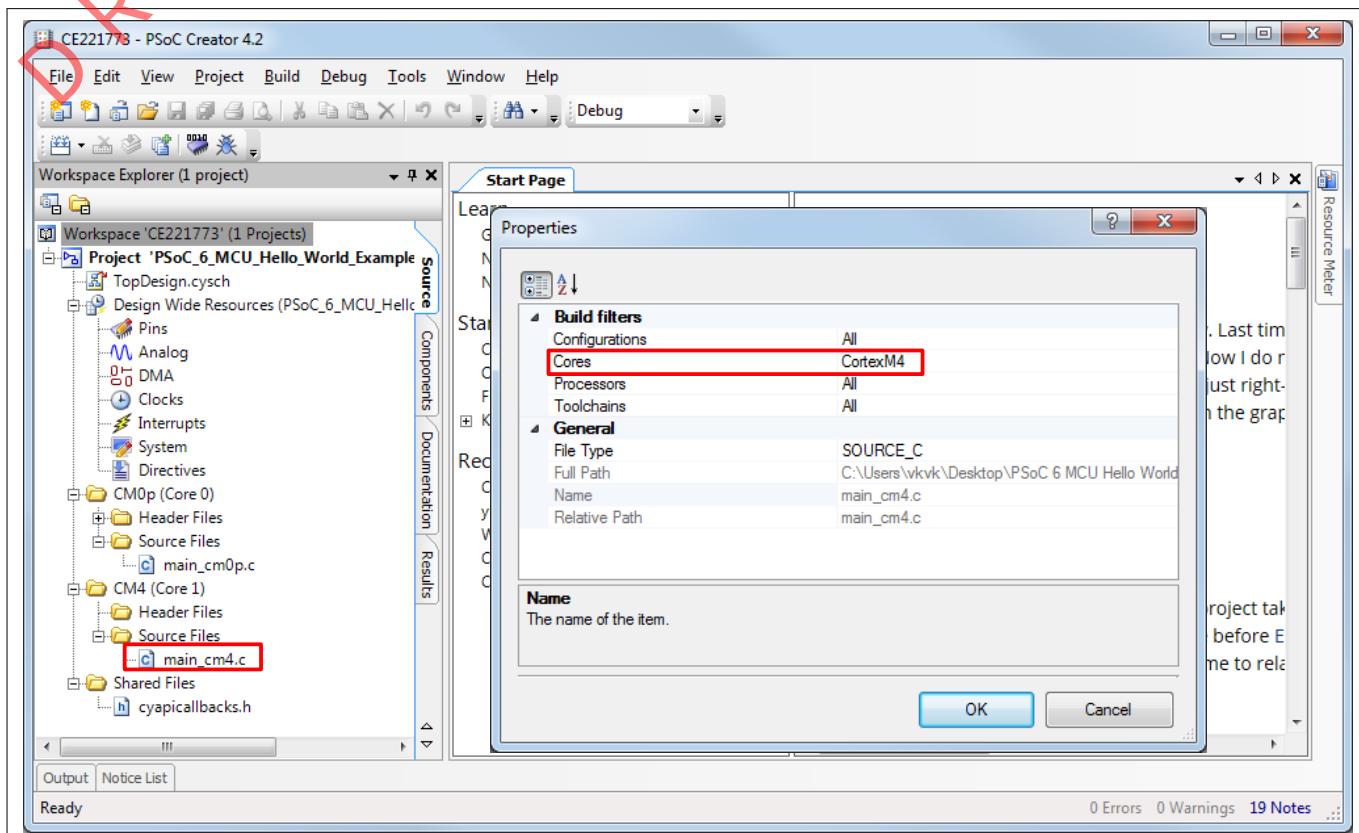


Figure 52 Setting target processor for a source C file

5.2.4.6 Part 4: Write the firmware

At this point in the development process, you have created a project, implemented a hardware design, and generated the code. In this part, you write the firmware that implements the design functionality.

The steps in this part discuss the firmware for the design that you configured in [Part 2: Implement the design](#).

The code example has all the required code. If you are working from scratch, you can copy the respective source codes to `main_cm0p.c` and `main_cm4.c` from the code snippet provided in this section. If you are using the code example, files are already in your project.

Firmware flow

In the remaining steps, we examine code in the `main_cm0p.c` and `main_cm4.c` file.

When the PSoC™ 6 MCU device is reset, the firmware first performs system initialization, which includes setting up the CPUs for execution, enabling global interrupts, and enabling other Components used in the design.

The initialization is split across the CPUs. The CM0+ CPU comes out of reset and enables the CM4 CPU. The CM0+ CPU code snippet is given in [Code listing 1](#). Copy the following code snippet to the `main_cm0p.c` file of your project.

5 PSoC™ 6 application notes

Code listing 1

```
/* Header files includes*/
#include "project.h"
int main(void)
{
    __enable_irq(); /* Enable global interrupts. */

    /* Enable CM4. CY_CORTEX_M4_APPL_ADDR must be updated
       if CM4 memory layout is changed. */
    Cy_SysEnableCM4(CY_CORTEX_M4_APPL_ADDR);

    for(;;)
    {

    }
}
/* [] END OF FILE */
```

When the CM4 CPU is enabled, the UART Component is started and prints a “Hello World!” message on the terminal emulator. A Timer Counter PWM (TCPWM) Component is configured to generate an interrupt every second. At each interrupt, the CM4 CPU toggles the LED (LED5) state on the kit. Copy the code snippet in [Code listing 2](#) to `main_cm4.c` of your project.

5 PSoC™ 6 application notes

Code listing 2

DRAFT

```
/* Header files includes*/
#include "project.h"

/********************* Macros ********************/
#define LED_ON      (0)
#define LED_OFF     (!LED_ON)

/********************* Function Prototypes ********************/
void UartInit(void);
void TimerInit(void);
void Isr_Timer(void);

/********************* Global Variables ********************/
bool LEDupdateFlag = false;

/********************* Function Name: main ********************/
int main(void)
{
    /* Start the UART peripheral */
    UartInit();

    /* Enable global interrupts. */
    __enable_irq();

    /* \x1b[2J\x1b[;H - ANSI ESC sequence for clear screen */
    Cy_SCB_UART_PutString(UART_HW, "\x1b[2J\x1b[;H");

    Cy_SCB_UART_PutString(UART_HW, "*****CE221773 - PSoC 6
MCU:\"
" Hello World! Example*****\r\n\r\n");

    Cy_SCB_UART_PutString(UART_HW, "Hello World!!!\r\n\r\n");

    Cy_SCB_UART_PutString(UART_HW, "Press Enter key to start blinking the
LED\r\n\r\n");

    /* Wait for the user to Press Enter */
    while(Cy_SCB_UART_Get(UART_HW) != '\r');

    /* Start the TCPWM peripheral. TCPWM is configured as a Timer */
    TimerInit();

    Cy_SCB_UART_PutString(UART_HW, "Observe the LED blinking on the kit!!!\r\n");
}
```

5 PSoC™ 6 application notes

DRAFT

```
for(;;)
{
    if(LEDupdateFlag)
    {
        /* Clear the flag */
        LEDupdateFlag = false;

        /* Invert the LED state*/
        Cy_GPIO_Inv(Pin_GreenLED_0_PORT, Pin_GreenLED_0_NUM);
    }
}
*****
* Function Name: UartInit
*****
void UartInit(void)
{
    /* Configure the UART peripheral.
     * UART_config structure is defined by the UART_PDL component based on
     * parameters entered in the Component configuration*/
    Cy_SCB_UART_Init(UART_HW, &UART_config, &UART_context);

    /* Enable the UART peripheral */
    Cy_SCB_UART_Enable(UART_HW);
}

*****
* Function Name: TimerInit
*****
void TimerInit(void)
{
    /* Configure the TCPWM peripheral.
     * Counter_config structure is defined based on the parameters entered
     * in the Component configuration */
    Cy_TCPWM_Counter_Init(Timer_HW, Timer_CNT_NUM, &Timer_config);

    /* Enable the initialized counter */
    Cy_TCPWM_Counter_Enable(Timer_HW, Timer_CNT_NUM);

    /* Start the enabled counter */
    Cy_TCPWM_TriggerStart(Timer_HW, Timer_CNT_MASK);

    /* Configure the ISR for the TCPWM peripheral*/
    Cy_SysInt_Init(&Isr_Timer_cfg, Isr_Timer);

    /* Enable interrupt in NVIC */
    NVIC_EnableIRQ((IRQn_Type)Isr_Timer_cfg.intrSrc);
}

*****
* Function Name: Isr_Timer
*****
```

5 PSoC™ 6 application notes

```

DRAFT
void Isr_Timer(void)
{
    /* Clear the TCPWM peripheral interrupt */
    Cy_TCPWM_ClearInterrupt(Timer_HW, Timer_CNT_NUM, CY_TCPWM_INT_ON_TC );

    /* Clear the CM4 NVIC pending interrupt for TCPWM */
    NVIC_ClearPendingIRQ(Isr_Timer_cfg.intrSrc);
    LEDupdateFlag = true;
}
/* [] END OF FILE */

```

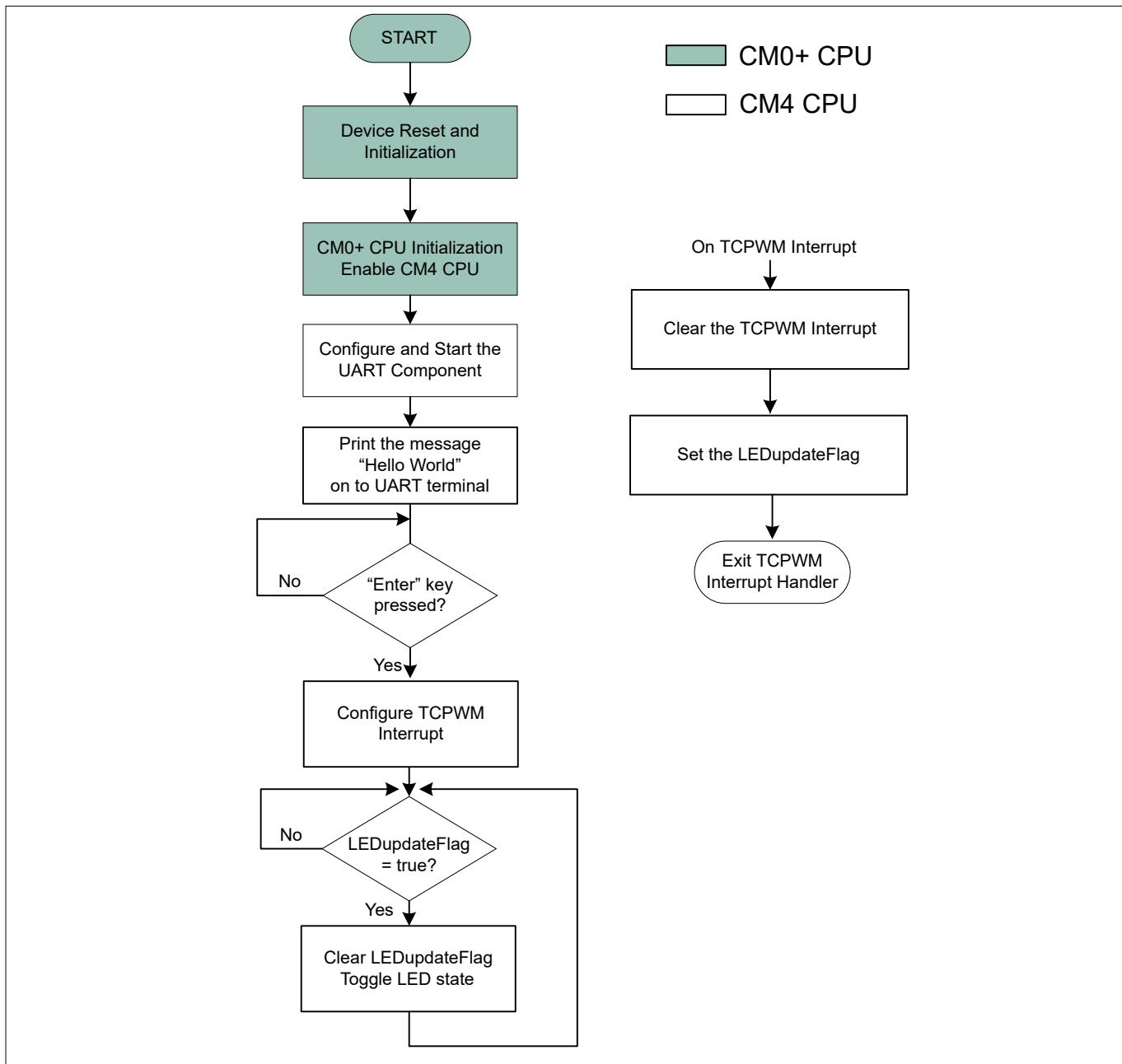


Figure 53 Firmware flowchart

This completes the summary of how the firmware works in the code example. Feel free to explore the source files for a deeper understanding.

~~5 PSoC™ 6 application notes~~

~~5.2.4.7 Part 5: Build the project and program the device~~

This section shows how to program the PSoC™ 6 MCU device. If you are using a development kit with a built-in programmer (the CY8CKIT-062-WiFi-BT Pioneer Kit, for example), connect the board to your computer using the USB cable. If you are developing on your own hardware, you may need a hardware programmer/debugger; for example, [CY8CKIT - 002 MiniProg3](#).

If you are working from scratch and encounter errors, revisit prior steps to ensure that you accomplished all the required tasks. You can work to resolve errors or switch to the code example for these final steps.

1. Select the debug target

PSoC™ Creator can debug one CPU at a time.

- In PSoC™ Creator, choose **Debug > Select Debug Target**, as Figure 54 shows

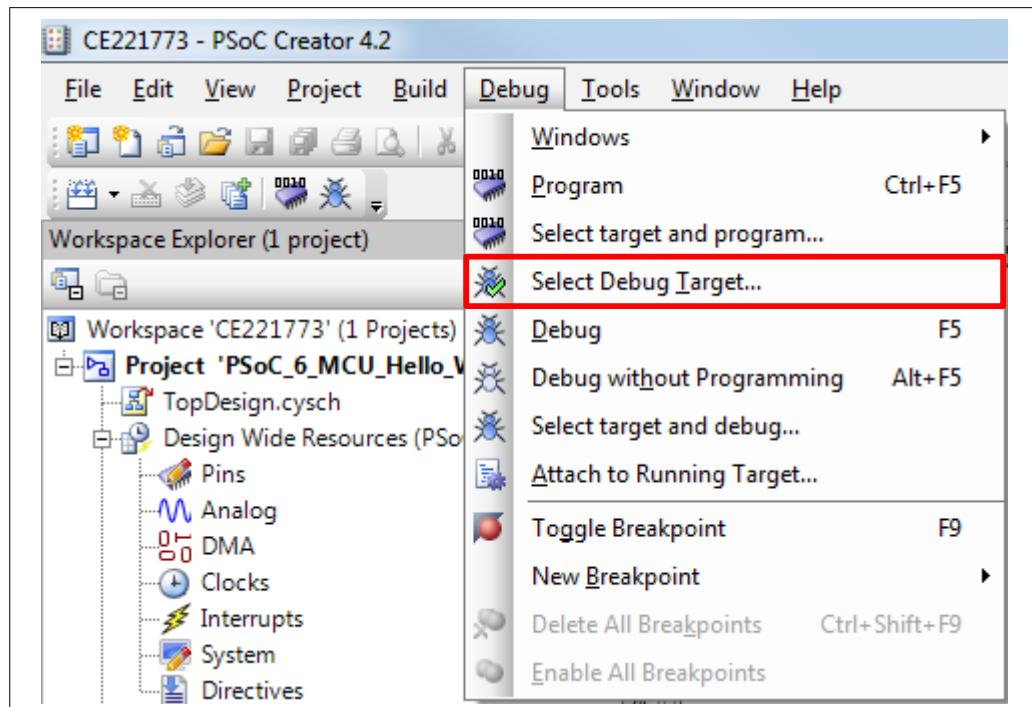


Figure 54

Selecting Debug Target

- Connect to the board

In the **Select Debug Target** dialog box, select the CM4 target, then click **OK/Connect**, as Figure 55 shows.

5 PSoC™ 6 application notes

DRAFT

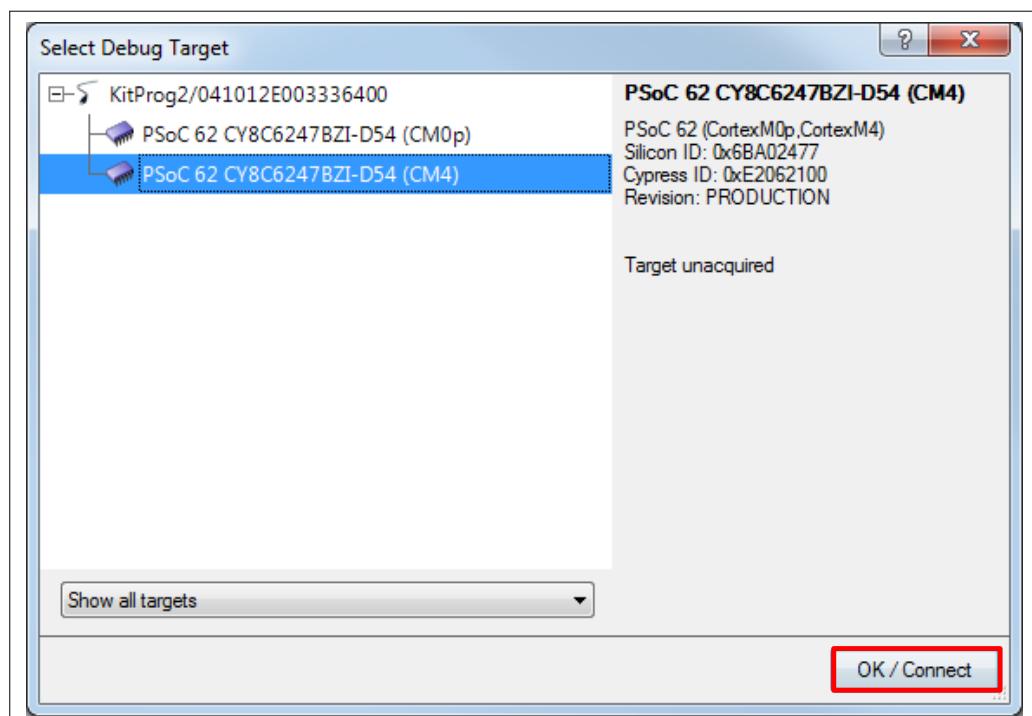


Figure 55 Connecting to a device

Note: For programming the board, you can pick either target. The CPUs share the same memory space. Programming either CPU programs both CPUs. However, if you are debugging, this choice matters. The debugger will see only the CPU you connect to. These instructions do not use the debugger

2. Program the board

Choose **Debug > Program** to program the device with the project, as [Figure 56](#) shows

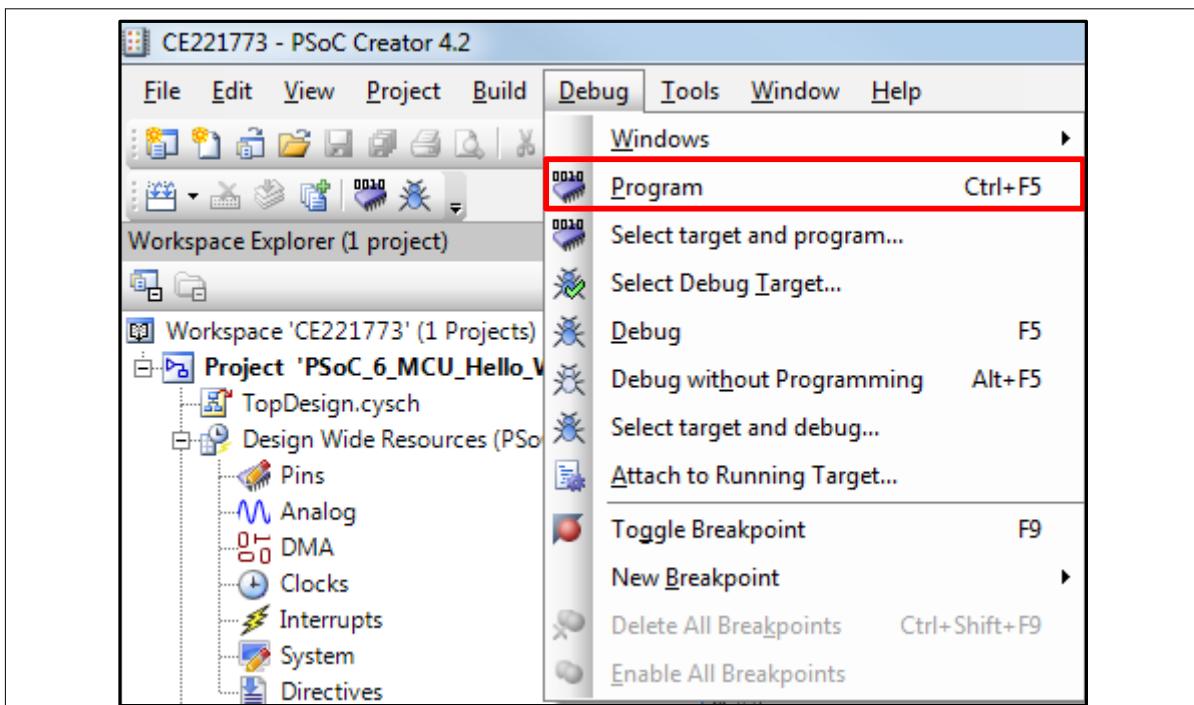


Figure 56 Programming the device

5 PSoC™ 6 application notes

You can view the programming status in the lower left corner of the PSoC™ Creator window, as Figure 57 shows.

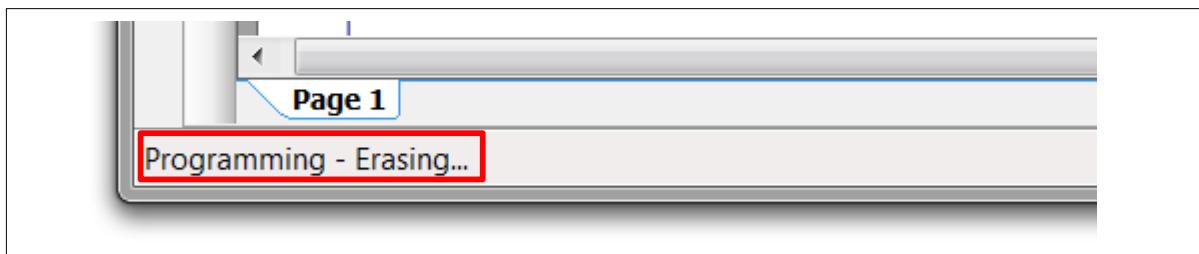


Figure 57 Programming status

Note: The **Debug > Debug** command also programs the board. If any code needs to be generated or rebuilt, that happens automatically when you issue a **Program** or **Debug** command. You can also debug without programming the board. However, these instructions do not use the debugger.

Note: The KitProg2 firmware on the kit might require an update. See the respective kit user guide for step-wise instructions on updating the firmware.

5.2.4.8 Part 6: Test your design

This section describes how to test your design.

Follow the steps below to observe the output of your design. Note that the below steps use Tera Term as the UART terminal emulator to view the results. You can use any terminal of your choice to view the output.

1. Select the serial port

Launch Tera Term and select the KitProg2 USB-UART COM port as shown in Figure 58.

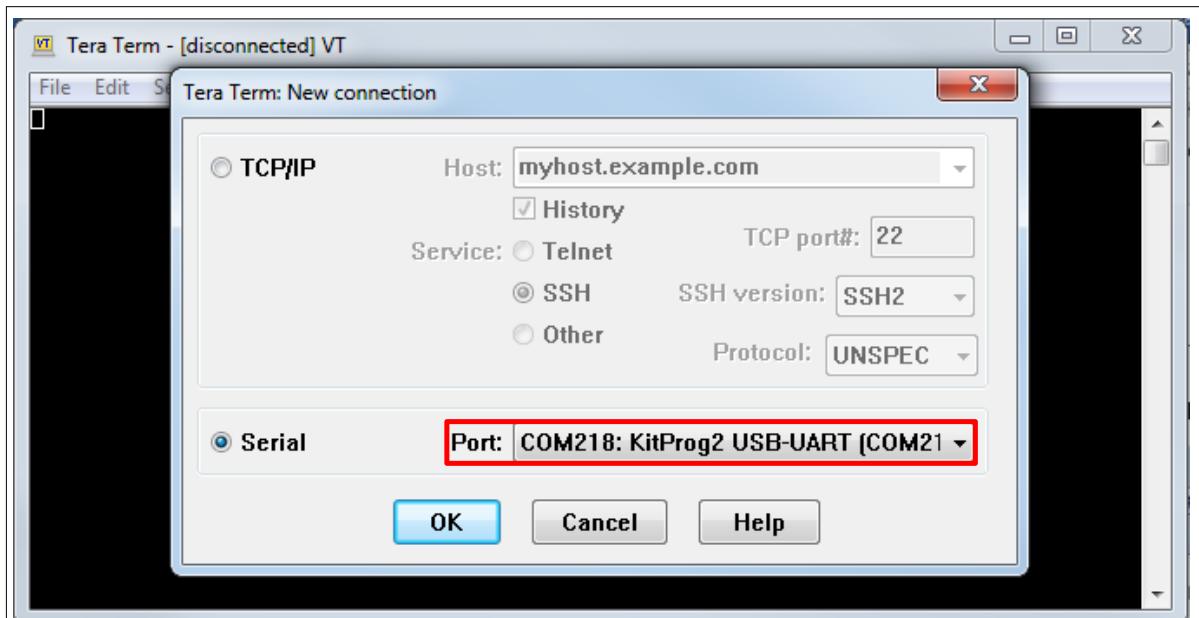


Figure 58 Selecting the KitProg2 USB-UART COM Port in Tera Term

2. Set the baud rate

Set the baud rate to 115200 under **Setup > Serial port** as Figure 59 shows

5 PSoC™ 6 application notes

DRAFT

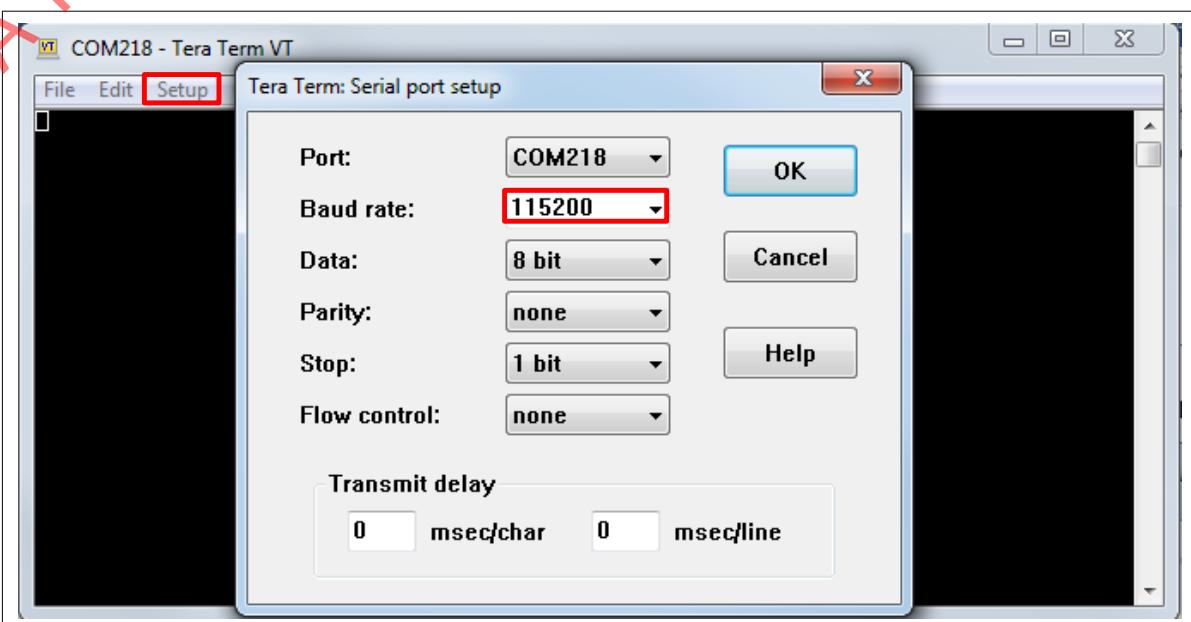


Figure 59 Configuring the Baud rate in Tera Term

3. Reset the device

Press the reset switch (SW1) on the Pioneer Kit. The following message appears on the terminal as [Figure 60](#) shows

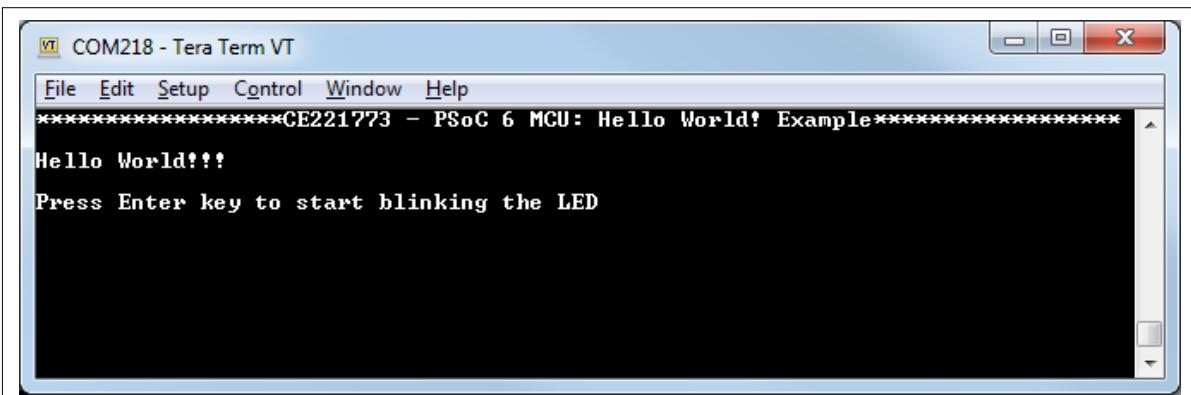


Figure 60 UART message printed from CM4 CPU

4. Enable the LED Blinking functionality

Press the **Enter** key to start blinking the LED. When the LED starts blinking, the following message will be displayed on the UART terminal as shown in [Figure 61](#).

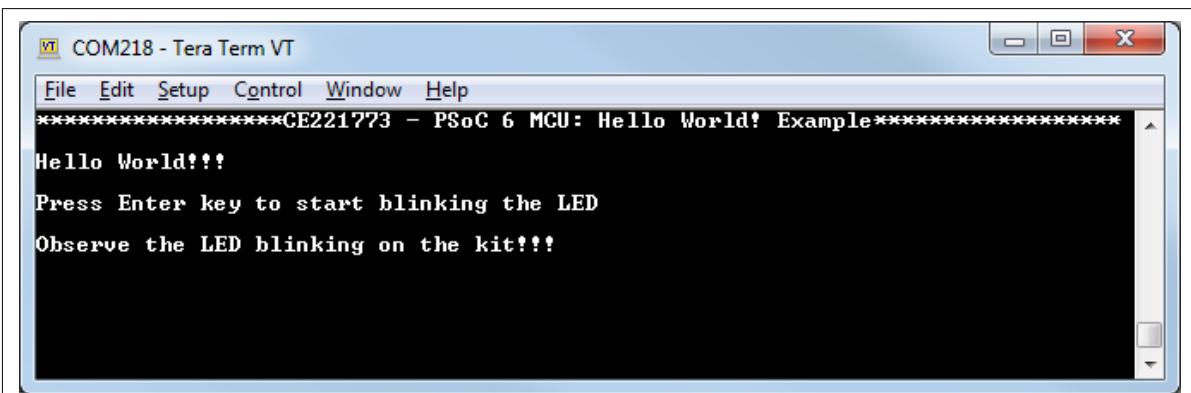


Figure 61 UART message from CM4 CPU

~~5 PSoC™ 6 application notes~~

~~5.2.5~~ Summary

This application note explored the PSoC™ 6 MCU device architecture and the associated development tools. PSoC™ 6 MCU is a truly programmable embedded system-on-chip with configurable analog and digital peripheral functions, memory, and a dual-CPU system on a single chip. The integrated features and low-power modes make PSoC™ 6 MCU an ideal choice for smart home, IoT gateways, and other related applications.

Related application notes and code examples

For a complete and updated list of PSoC™ 6 MCU code examples, please visit our [code examples web page](#). For more PSoC™ 6 MCU-related documents, please visit our [PSoC™ 6 MCU product web page](#).

[Table 6](#) lists the system-level and general application notes that are recommended for the next steps in learning about PSoC™ 6 MCU and PSoC™ Creator.

Table 6 General and system-level application notes

Document	Document name
AN228571	Getting Started with PSoC™ 6 MCU on ModusToolbox
AN210781	Getting Started with PSoC™ 6 MCU with Bluetooth™ Low Energy (BLE) Connectivity on PSoC™ Creator
AN218241	PSoC™ 6 MCU Hardware Design Considerations
AN219434	PSoC™ 6 MCU Importing Generated Code into an IDE
AN219528	PSoC™ 6 MCU Low-Power Modes and Power Reduction Techniques

[Table 7](#) lists the application notes (AN) and code examples (CE) for specific peripherals and applications.

Table 7 Documents related to PSoC™ 6 MCU features

Document	Document name
System resources, CPU, and interrupts	
AN215656	PSoC™ 6 MCU Dual-CPU System Design
AN217666	PSoC™ 6 MCU Interrupts
CE221773	PSoC™ 6 MCU Hello World Example
CE216795	PSoC™ 6 MCU Dual-Core Basics
CE216825	PSoC™ 6 MCU Real-Time Clock Basics
CE218129	PSoC™ 6 MCU Wake up from Hibernate Using Low-Power Comparator
CE218541	PSoC™ 6 MCU Fault-Handling Basics
CE218542	PSoC™ 6 Custom Tick Timer Using RTC Alarm Interrupt
CE218552	PSoC™ 6 MCU UART to Memory Buffer Using DMA
CE218964	PSoC™ 6 MCU RTC Daily Alarm
CE219339	PSoC™ 6 MCU MCWDT and RTC Interrupts (Dual Core)
CE219521	PSoC™ 6 MCU GPIO Interrupt
CE219881	PSoC™ 6 MCU Switching Power Modes
CE220060	PSoC™ 6 MCU Watchdog Timer
CE220061	PSoC™ 6 MCU Multi-Counter Watchdog Interrupts
CE220120	PSoC™ 6 MCU Blocking Mode Flash Write
CE220169	PSoC™ 6 MCU Periodic Interrupt Using TCPWM

(table continues...)

Reference manual

~~DRAFT~~

5 PSoC™ 6 application notes

Table 7 (continued) Documents related to PSoC™ 6 MCU features

Document	Document name
GPIO	
CE219490	PSoC™ 6 Breathing LED Using SMART IO
CE219506	PSoC™ 6 Clock Buffer Using SMART IO
CE220263	PSoC™ 6 MCU GPIO Pins Example
CAPSENSE™	
AN92239	Proximity Sensing with CapSense
AN85951	PSoC™ 4 and PSoC™ 6 MCU CapSense Design Guide
Bootloader	
AN213924	PSoC™ 6 MCU Bootloader Software Development Kit (SDK) Guide
CE213903	PSoC™ 6 MCU Basic Bootloaders
Communications	
CE220541	PSoC™ 6 MCU SCB EzI2C
Audio	
CE218636	PSoC™ 6 MCU Inter-IC Sound (I2S) Example
CE219431	PSoC™ 6 MCU PDM-to-PCM Example
RTOS	
CE217911	PSoC™ 6 MCU FreeRTOS™ Example Project
Security	
CE220465	PSoC™ 6 MCU Cryptography – AES Demonstration
CE220511	PSoC™ 6 MCU Cryptography – SHA Demonstration

5 PSoC™ 6 application notes

Glossary

2

This section lists the most commonly used terms that you might encounter while working with PSoC™ family of devices.

- **Component Customizer:** Simple GUI in PSoC™ Creator that is embedded in each Component. It is used to customize the Component parameters and is accessed by right-clicking a Component
- **Components:** Components are used to integrate multiple ICs and system interfaces into one PSoC™ Component that is inherently connected to the MCU via the main system bus. For example, the Bluetooth® LE Component creates Bluetooth® Smart products in minutes. Similarly, you can use the Programmable Analog Components for sensors
- **KitProg:** The KitProg is an onboard programmer/debugger with USB-I2C and USB-UART bridge functionality. The KitProg is integrated onto most PSoC™ development kits
- **MiniProg3/MiniProg4:** Programming hardware for development that is used to program PSoC™ devices on your custom board or PSoC™ development kits that do not support a built-in programmer
- **Personality:** A personality expresses the configurability of a resource for a functionality. For example, the SCB resource can be configured to be an UART, SPI or I2C personalities
- **PSoC™:** A programmable, embedded design platform that includes a CPU, such as the 32-bit Arm Cortex-M0, with both analog and digital programmable blocks. It accelerates embedded system design with reliable, easy-to-use solutions, such as touch sensing, and enables low-power designs
- **ModusToolbox™:** An Eclipse based embedded design platform for IoT designers that provides a single, coherent, and familiar design experience combining the industry's most deployed Wi-Fi and Bluetooth® technologies, and the lowest power, most flexible MCUs with best-in-class sensing
- **PSoC™ Creator:** PSoC™ 3, PSoC™ 4, PSoC™ 5LP, PSoC™ 6 MCU, and PSoC™ 6 Bluetooth® LE Integrated Design Environment (IDE) software that installs on your PC and allows concurrent hardware and firmware design of PSoC™ systems, or hardware design followed by export to other popular IDEs
- **Peripheral Driver Library:** The Peripheral Driver Library (PDL) simplifies software development for the PSoC™ 6 MCU architecture. The PDL reduces the need to understand register usage and bit structures, so easing software development for the extensive set of peripherals available
- **PSoC™ Programmer:** A flexible, integrated programming application for programming PSoC™ devices. PSoC™ Programmer is integrated with PSoC™ Creator to program PSoC™ 3, PSoC™ 4, PRoC, PSoC™ 5LP, PSoC™ 6 MCU, and PSoC™ 6 Bluetooth® LE designs
- **WICED:** WICED (Wireless Internet Connectivity for Embedded Devices) is a full-featured platform with proven Software Development Kits (SDKs) and turnkey hardware solutions from partners to readily enable Wi-Fi and Bluetooth® connectivity in system design

~~5 PSoC™ 6 application notes~~

~~5.2.6 Revision history~~

Document version	Date of release	Description of changes
**	2018-03-28	New application note
*A	2018-10-04	Updated for ModusToolbox™
*B	2018-10-31	Updated images
*C	2018-11-15	Updated for public release
*D	2019-02-19	Updated for ModusToolbox™ 1.1
*E	2019-10-14	Removed content for ModusToolbox™ 1.1
*F	2021-03-14	Updated to Infineon Template
*G	2022-07-21	Template update

5.3 AN210781 Getting started with PSoC™ 6 MCU with Bluetooth® low energy connectivity on PSoC™ Creator

About this document

-
- 3

Scope and purpose

AN210781 introduces you to PSoC™ 6 MCU with Bluetooth® Low Energy connectivity, a dual-CPU Arm® Cortex®-M4 and Cortex®-M0+ based programmable system-on-chip that integrates a Bluetooth® Low Energy 5.0 system, the latest-generation of CAPSENSE™ technology, and a host of security features. This application note helps you explore the PSoC™ 6 MCU with Bluetooth® Low Energy architecture and development tool and shows you how to create your first project using PSoC™ Creator, export the project to a third-party integrated development environment (IDE), and continue your firmware development. It also guides you to more resources available online to accelerate your learning about PSoC™ 6 MCU with Bluetooth® Low Energy connectivity. To get started with the PSoC™ 6 MCU device family, see [AN221774 – Getting started with PSoC™ 6 MCU](#).

To access an ever-growing list of hundreds of PSoC™ code examples, please visit our [code examples web page](#). You can also explore the PSoC™ video library [here](#).

~~DEAF~~ 5 PSoC™ 6 application notes

5.3.1 Introduction

PSoC™ 6 MCU with Bluetooth® LE connectivity, hereafter called as PSoC™ 6-BLE, is an ultra-low-power PSoC™ device specifically designed for wearables and Internet of Things (IoT) products. It establishes a new low-power standard for today's "always-on" applications. The PSoC™ 6 -BLE device is a programmable embedded system-on-chip that integrates the following on a single chip:

- Dual-CPU microcontroller: CM4 and CM0+
- Bluetooth® LE 5.0 subsystem
- Programmable analog and digital peripherals
- Up to 1 MB of flash and 288 KB of SRAM
- Fourth-generation CAPSENSE™ technology

PSoC™ 6-BLE is suitable for a variety of power-sensitive connected applications such as:

- Smart watches and fitness trackers
- Connected medical devices
- Smart home sensors and controllers
- Smart home appliances
- Gaming controllers
- Sports, smart phone, and virtual reality (VR) accessories
- Industrial sensor nodes
- Industrial logic controllers
- Advanced remote controllers

PSoC™ 6-BLE provides a cost-effective and small-footprint alternative to the combination of an MCU and a Bluetooth® LE radio. The programmable analog and digital subsystems allow flexibility and dynamic fine-tuning of the design using PSoC™ Creator the schematic-based design tool for developing PSoC™ 6-BLE applications. To develop a BLE application, you do not need a working knowledge of the complex BLE protocol stack.

PSoC™ 6-BLE is an easy-to-configure, no-cost GUI-based Bluetooth® LE component in PSoC™ Creator that abstracts the protocol complexity.

Bluetooth® LE is an ultra-low-power wireless standard defined by the Bluetooth® Special Interest Group (SIG) for short-range communication. PSoC™ 6-BLE integrates a Bluetooth® LE 4.2 radio and a royalty-free protocol stack with enhanced security, privacy, and throughput compliant with the BLE 5.0 specification.

Fourth-generation capacitive touch-sensing feature in PSoC™ 6-BLE devices, known as CAPSENSE™, offers unprecedented signal-to-noise ratio (SNR); best-in-class waterproofing; and a wide variety of sensor types such as buttons, sliders, track pads, and proximity sensors. CAPSENSE™ user interfaces are gaining popularity in wearable electronic devices such as activity monitors and health and fitness equipment. The CAPSENSE™ solution works in noisy environments and in the presence of liquids.

PSoC™ 6-BLE enables ultra-low-power connected applications with an integrated solution.

Figure 62 shows the application-level block diagram of a fitness tracker based on PSoC™ 6-BLE.

5 PSoC™ 6 application notes

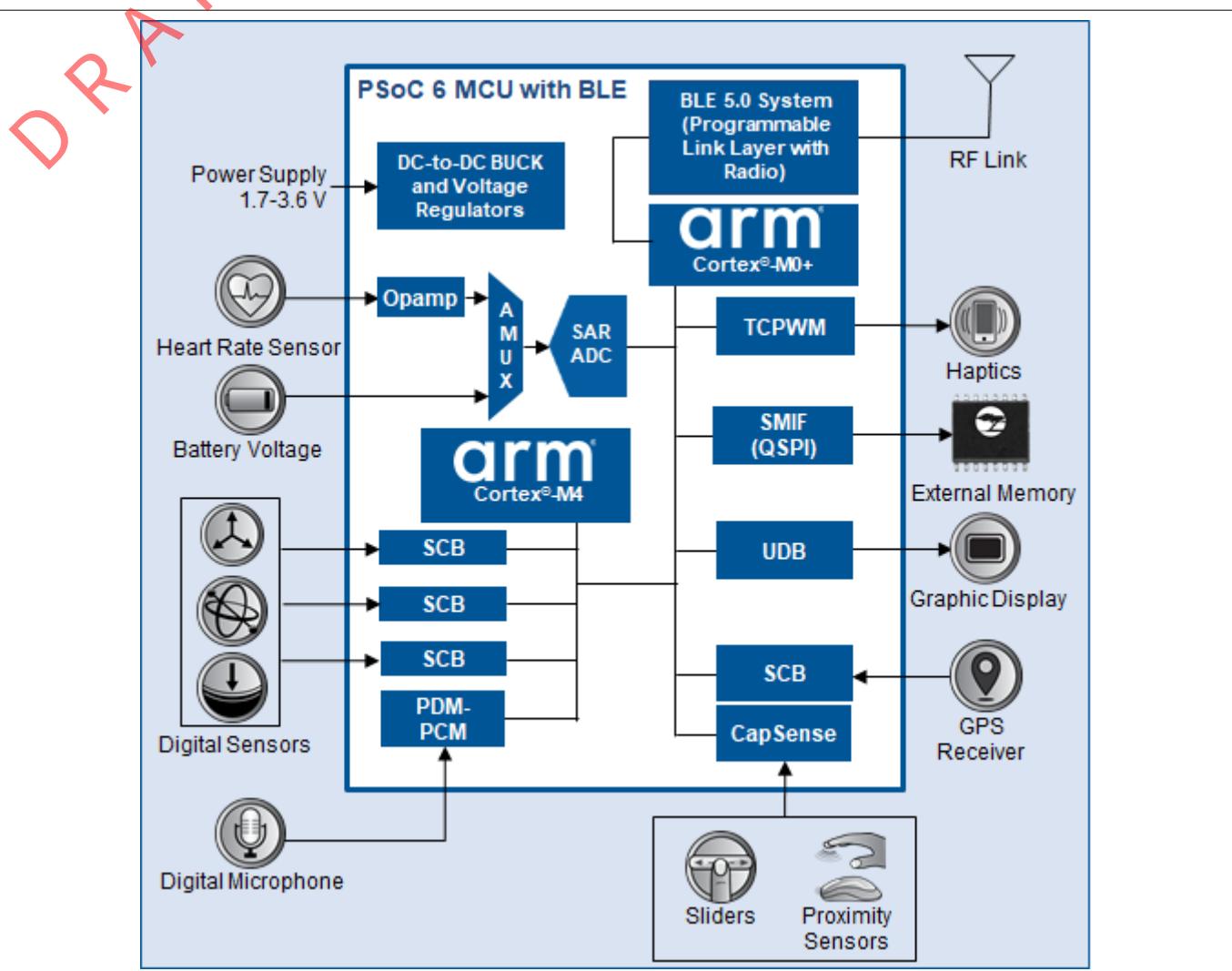


Figure 62 Fitness tracker application block diagram

The device provides a one-chip solution and includes:

- A low-power Bluetooth® LE 5.0 system that can sustain up to four simultaneous connections
- A buck converter for ultra-low-power operation
- An analog front end (AFE) within the device to condition and measure heart rate sensor outputs and to monitor battery voltage
- Serial communication blocks (SCBs) to interface with multiple digital sensors including a global positioning system (GPS) module
- A Pulse-Density Modulation (PDM) Pulse Code Modulation (PCM) hardware engine and digital microphone interface for voice
- CAPSENSE™ technology for reliable touch and proximity sensing
- A serial memory interface (SMIF) that supports an interface to Quad-Serial Peripheral Interface (QSPI)-enabled external memory
- Digital logic (UDB) and peripherals (TCPWM) to drive the display and haptics

See [Device features](#) and [Appendix C](#) for more details on device features.

This application note introduces you to the capabilities of PSoC™ 6-BLE, gives an overview of the development ecosystem, and gets you started with a simple design: a Bluetooth® SIG standard [Find Me Profile \(FMP\)](#) with

5 PSoC™ 6 application notes

~~SECRET~~
an [Immediate Alert Service \(IAS\)](#). This design is available as code example [CE212736](#) for [CY8CKIT-062-BLE](#). This application note uses the code example extensively.

For hardware design considerations, see [AN218241 – PSoC™ 6 MCU hardware design considerations](#).

For advanced application development, refer to the application note [AN215796 – Designing PSoC™ 6-BLE applications](#).

5.3.1.1 Prerequisites

Before you get started, make sure you have the development kit and have installed the required software, including the code example.

5.3.1.1.1 Hardware

- [CY8CKIT-062-BLE PSoC™ 6-BLE pioneer kit](#)
- PC with Windows 7 or later (if using the [CySmart™ Host Emulation Tool](#) application)
- Mobile phone with Android 5/iOS 8 or later (if using the [CySmart™ iOS/Android app](#))

5.3.1.1.2 Software

- [PSoC™ Creator 4.4 with PSoC™ Programmer 3.29](#)
- [CE212736](#) – the Bluetooth® LE Find Me code example for the [CY8CKIT-062-BLE PSoC™ 6-BLE pioneer kit](#)
- [CySmart™ Host Emulation Tool](#) PC application or [CySmart™ iOS/Android app](#)

Scan the following QR codes from your mobile phone to download the CySmart app.



iOS



Android

5 PSoC™ 6 application notes

5.3.2 Development ecosystem

5.3.2.1 PSoC™ resources

A wealth of data is available at www.cypress.com to help you to select the right PSoC™ device and quickly and effectively integrate it into your design. The following is an abbreviated list of resources for PSoC™ 6 MCU:

- **Overview:** [PSoC™ Portfolio](#), [PSoC™ Roadmap](#)
- **Product selectors** [PSoC™ 6 MCU](#)
- [Datasheets](#) describe and provide electrical specifications for each device family
- [Application notes](#) and [Code examples](#) cover a broad range of topics, from basic to advanced level. Many of the application notes include code examples. You can also browse our collection of code examples from directly inside PSoC™ Creator — see [Code examples](#)
- [Technical reference manuals \(TRMs\)](#) provide detailed descriptions of the architecture and registers in each device family
- [PSoC™ 6 programming specification](#) provide the information necessary to program the nonvolatile memory of the PSoC™ 6 MCU devices
- [CAPSENSE™ design guides](#): Learn how to design capacitive touch-sensing applications with PSoC™ devices
- **Development Tools**
- [CY8CKIT-062-WiFi-BT PSoC™ 6 WiFi-BT pioneer kit](#) is a development kit that supports the PSoC™ 62 series MCU along with WiFi and Bluetooth® connectivity
- [CY8CKIT-062-BLE PSoC™ 6-BLE pioneer kit](#) is an easy-to-use and inexpensive development platform for PSoC™ 6-BLE
- **Training videos:** [Video training](#) on our products and tools, including a dedicated series on [PSoC™ 6 MCU](#)

For a comprehensive list of PSoC™ 6 MCU resources, see [KBA223067](#) in the Infineon community.

5.3.2.2 Firmware/application development

PSoC™ Creator and the Peripheral driver library (PDL) are at the heart of the development process.

PSoC™ Creator brings together several digital/analog/system components and firmware to build an application. Using PSoC™ Creator, you can select, place, and configure components on a schematic; write C/assembly source code; and program and debug the device.

The PDL is the software development kit for the PSoC™ 6 MCU family. The PDL, provided as source code, makes it easier to develop the firmware for supported devices. It helps you quickly customize and build firmware without the need to understand the register set. The PDL supports both PSoC™ Creator and third-party IDEs.

5.3.2.2.1 PSoC™ Creator

[PSoC™ Creator](#) is a free Windows-based Integrated Design Environment (IDE). It enables you to design hardware and firmware systems concurrently, based on PSoC™ 6 MCU. As [Figure 63](#) shows, with PSoC™ Creator, you can

Steps

1. Browse collection of code examples from the **File > Code Example...** menu.
 - a. Filter for examples based on Device family.
 - b. Select from the menu of examples offered based on the **Filter by** options.
 - c. Download the code example using the download button.
 - d. Create a new project based on the selection.
2. Explore the library of more than 100 components.

5 PSoC™ 6 application notes

- ~~DRAFT~~
3. Drag and drop components to build your hardware system design in the main design workspace.
 4. Review the component datasheets.
 5. Configure the components using configuration tools.
 6. Co-design your application firmware with the PSoC™ hardware.

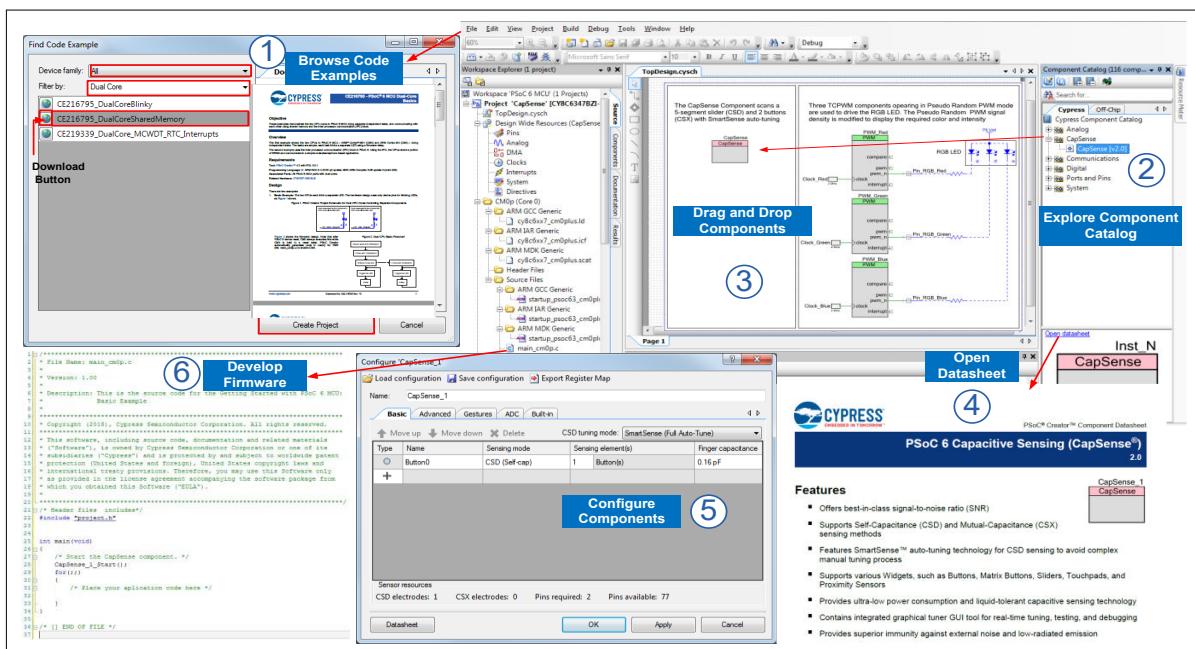


Figure 63

PSoC™ Creator schematic entry and components

PSoC™ Creator help

Visit the [PSoC™ Creator](#) home page to download and install the latest version of PSoC™ Creator. Then launch PSoC™ Creator and navigate to the following items:

- **Quick start guides:** Choose **Help > Documentation > Quick Start Guide**. This guide gives you the basics for developing PSoC™ Creator projects
- **Code examples:** Choose **File > Code Example** or click the **Find Code Example...** link on the **Start Page** tab. These code examples demonstrate how to configure and use PSoC™ resources
- **Component datasheets:** Right-click a Component and select **Open Datasheet**. Visit the [PSoC™ 6 MCU Component Datasheets](#) page for a list of all Component datasheets

5.3.2.2 Peripheral Driver Library (PDL)

The PDL is the software development kit for PSoC™ 6 MCU devices. If you have experience working with PSoC™ devices but are new to PSoC™ 6 MCU, you will notice that PDL is the major addition to the development ecosystem.

Firmware developers who wish to work at the register level should also install the PDL. The PDL includes all the device-specific header files and startup code you need for your project. It also serves as a reference for each driver. Because the PDL is provided as source code, you can see how it accesses the hardware at the register level.

Some devices do not support particular peripherals. The PDL is a superset of all drivers for any supported device. This superset design means:

- All APIs needed to initialize, configure, and use a peripheral are available

5 PSoC™ 6 application notes

- ~~CONFIDENTIAL~~
- The PDL is useful across all PSoC™ 6 MCU devices, regardless of available peripherals
 - The PDL includes error checking to ensure that the targeted peripheral is present on the selected device

This enables the code to maintain compatibility across platform as long as the peripherals are available. A device header file specifies the peripherals that are available for a device. If you write code that attempts to use an unsupported peripheral, you will get an error at compile time. Before writing code to use a peripheral, consult the datasheet for the particular device to confirm support for the peripheral.

PSoC™ Creator provides components that are based on PDL for your use with PSoC™ 6 MCU devices. This retains the essence of PSoC™ Creator in utilizing community-developed and pre-validated Components. However, the PDL is a source code library that you can use with any development environment.

The PDL includes the following key software resources:

- Header and source files for each peripheral driver
- Header and source files for middleware libraries
- Device-specific header, startup, and configuration files
- Template projects for supported third-party IDEs
- Full documentation

The location for the documentation is `<PDL install directory>|doc`. There are two key documents.

The PDL v3.x user guide covers the fundamentals of working with the PDL, such as:

- Creating a custom project using the PDL (including third-party IDEs)
- Configuring a peripheral
- Managing pins in firmware
- Using the PDL as a learning tool for register-based programming
- Using the PDL API Reference documentation

The second key document is the PDL 3.x API reference manual.html. This reference has complete information on every driver in the PDL, including overview, configuration considerations, and details on every function, macro, data structure, and enumerated type

5.3.2.3 Support for other IDEs

You can also develop firmware for PSoC™ 6 MCU using your favorite IDE such as [IAR Embedded Workbench](#). You can use the PDL with another IDE by using PSoC™ Creator to design the system and generate configuration code and then export to a target IDE.

See the [AN219434 – PSoC™ 6 MCU Importing Generated Code into an IDE](#) for details.

5.3.2.3.1 Using PSoC™ Creator to target another IDE

PSoC™ Creator sets up and configure PSoC™ 6 MCU system resources and peripherals. You then export the project to your IDE and continue developing firmware in your IDE. If there is a change in the device configuration, you edit the Topdesign schematic in PSoC™ Creator and regenerate the code for the target IDE. Figure 64 shows the workflow.

5 PSoC™ 6 application notes

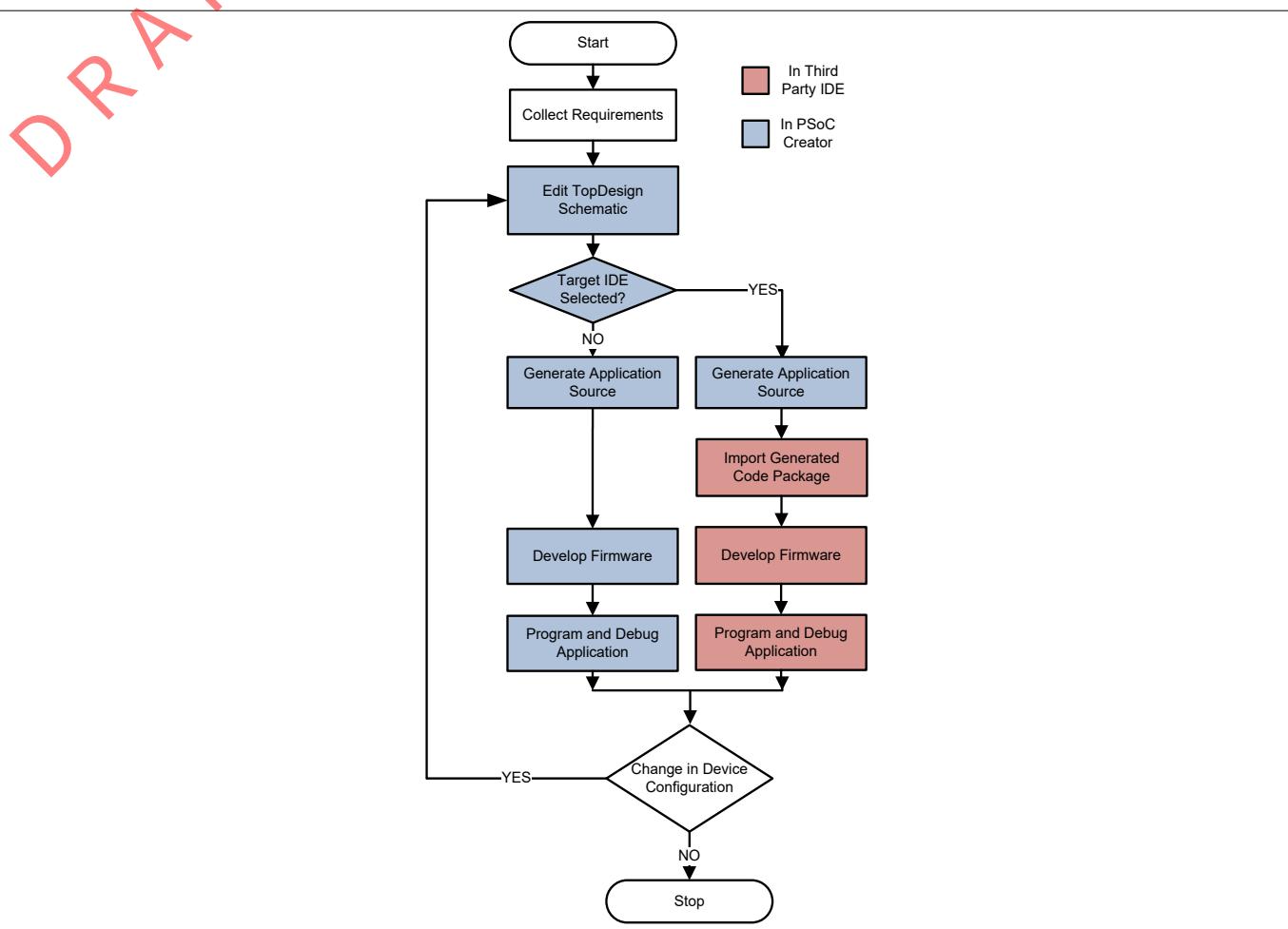


Figure 64 PSoC™ 6 MCU application development flowchart using PSoC™ Creator

Code generated from PSoC™ Creator includes all required header, startup, and PDL files based on your design. Exporting the code generated from PSoC™ Creator is supported for Keil µVision, IAR Embedded Workbench, Eclipse-based IDEs, and for GNU-based command-line tools. You select your target IDE using the **Project > Build Settings > Target IDEs** panel. Enable the export package for your target IDE as shown in [Figure 65](#). When you generate code, PSoC™ Creator also creates the corresponding export package.

5 PSoC™ 6 application notes

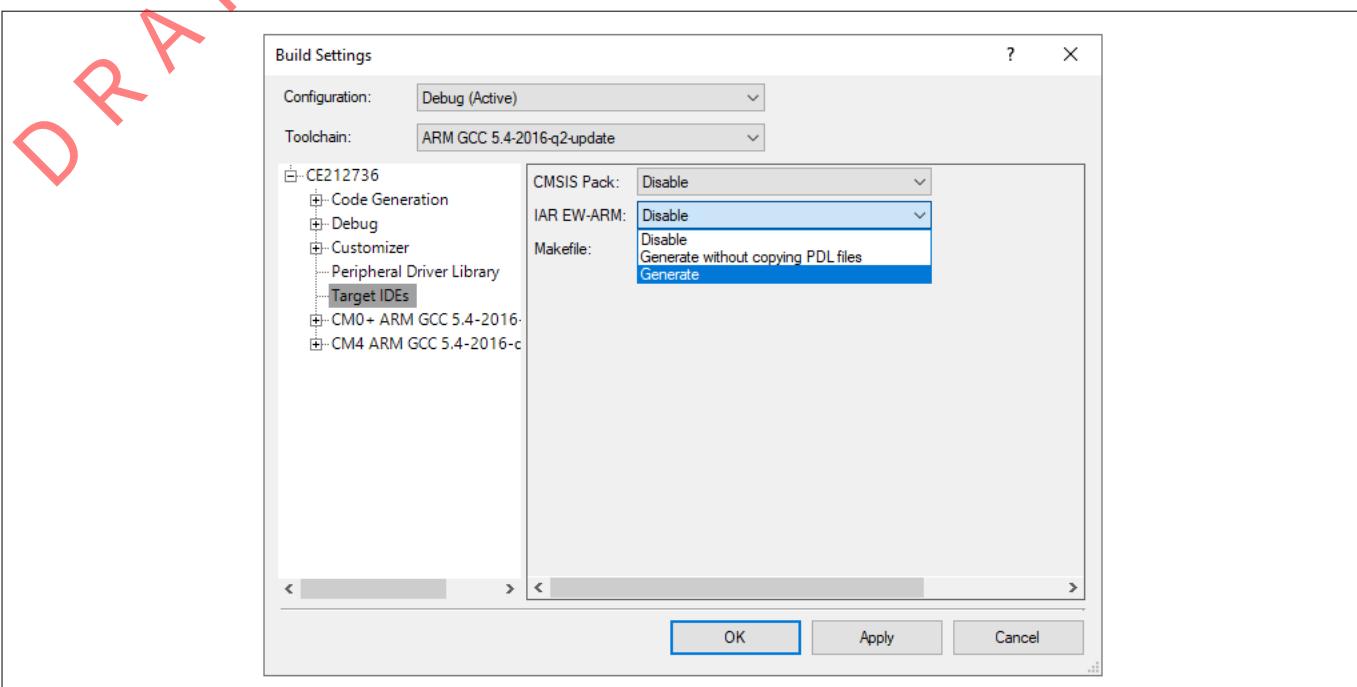


Figure 65 Target IDE selection in project > Build settings

You then import the package into your IDE. The import details vary significantly per IDE. Consult the PSoC™ Creator Help to learn the process you must follow. Choose **Help > PSoC™ Creator Help Topics**. See [Figure 66](#). The *Integrating into 3rd Party IDEs* topic points you to specific help files for each PSoC™ device family, and each supported IDE.

The screenshot shows the 'Welcome to PSoC Creator' help page for Release 4.4. A red arrow points to the 'Integrating into 3rd Party IDEs' link in the table of contents.

Getting Started	Tutorials to get you started using PSoC Creator.
Understanding PSoC Creator	Information and tasks to better understand PSoC Creator.
Using Design Entry Tools	Tasks and interface descriptions for the graphical design entry tools.
Building a PSoC Creator Project	Topics for configuring and building PSoC Creator projects.
Integrating into 3rd Party IDEs	Information and tasks for generating PSoC Creator design files for use with a 3rd Party IDE
Programming and Debugging a PSoC Creator Project	Topics for programming the device and using the debugger.
Completing the Project	Topics for finalizing a PSoC Creator design.
Reference Material	3rd party tool chain docs and other reference material.

Figure 66 PSoC™ Creator help

You can work effectively in most if not all IDEs. If your IDE is not supported in the Target IDEs panel, you can still use PSoC™ Creator. After you generate code, add the necessary files directly to your IDE's project. [AN219434 – PSoC™ 6 MCU Importing generated code into an IDE](#) provide detailed steps for manually importing the generated code into another IDE.

~~5 PSoC™ 6 application notes~~

~~5.3.2.4 RTOS support~~

The PDL includes RTOS support for PSoC™ 6 MCU development: FreeRTOS source code is fully integrated and included with the PDL. You can import the FreeRTOS software package into your project by using the PSoC™ Creator RTOS import option. Navigate to **Project > Build Settings** menu and select FreeRTOS from the Software package imports option under **Peripheral Driver Library > FreeRTOS** as shown in [Figure 67](#).

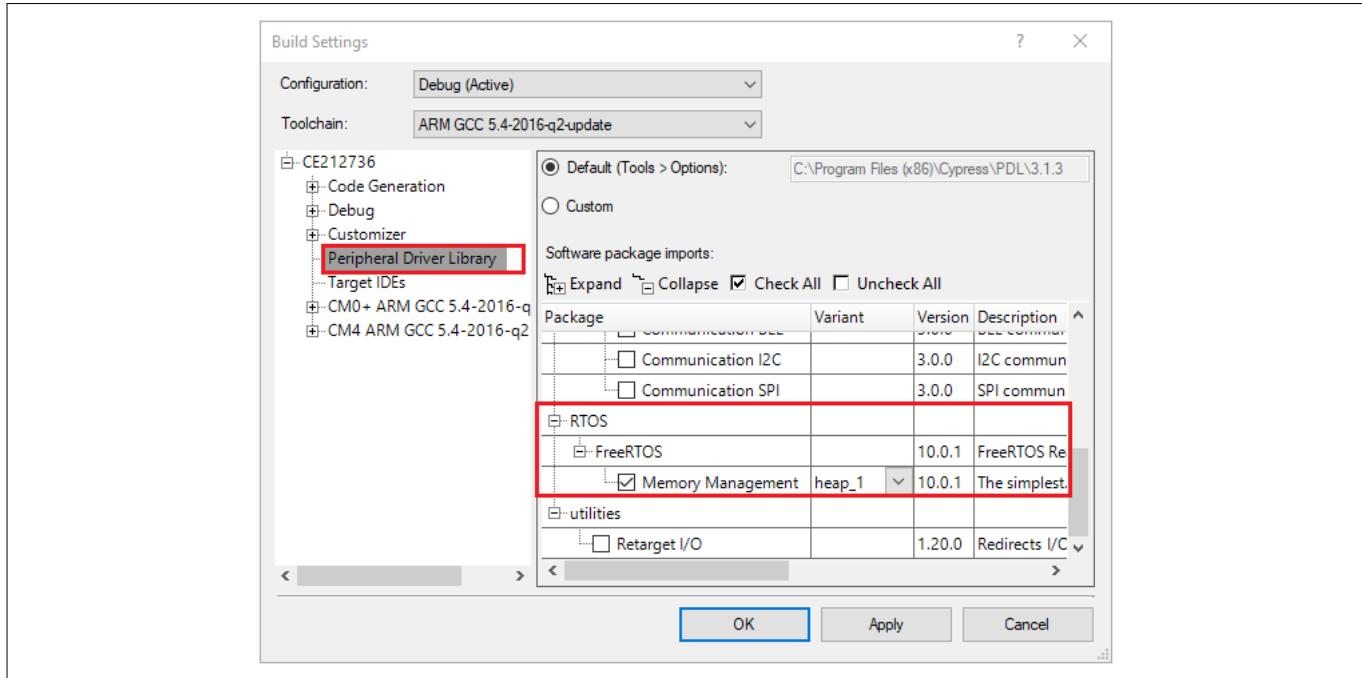


Figure 67 Import FreeRTOS in PSoC™ Creator project

If you have a preferred RTOS, use the resources provided as examples on how to integrate such code with the PDL.

5.3.2.5 Debugging

The [CY8CKIT-062-BLE PSoC™ 6-BLE pioneer kit](#) has the KitProg2 onboard programmer/debugger.

Note: *Update the onboard programmer/debugger to KitProg3 using PSoC™ Programmer 3.29. KitProg3 uses industry-standard CMSIS-DAP as the transport mechanism. KitProg3 implements USB Bulk endpoints for faster communication. It also supports HID endpoints for use cases that require them, but communication is slower. Out of the box, KitProg3 uses Bulk endpoints. KitProg3 also supports bridging: USB-UART; USB-I2C, and USB-SPI. This makes debugging the PSoC™ 6 MCU pioneer kit extremely flexible. See the [KitProg3 User Guide](#) for details.*

PSoC™ Creator supports debugging a single CPU (either CM4 or CM0+) at a time. Some third-party IDEs support multi-CPU debugging. For more information on debugging PSoC™ devices with PSoC™ Creator, see the PSoC™ Creator help. To update onboard programmer/debugger to KitProg3, plug in your kit and open.

PSoC™ Programmer 3.29 tool. The tool automatically detects an older version of KitProg2 or KitProg3 on the kit and offers to upgrade the KitProg firmware to a newer version of KitProg3 as shown in [Figure 68](#) and [Figure 69](#).

5 PSoC™ 6 application notes

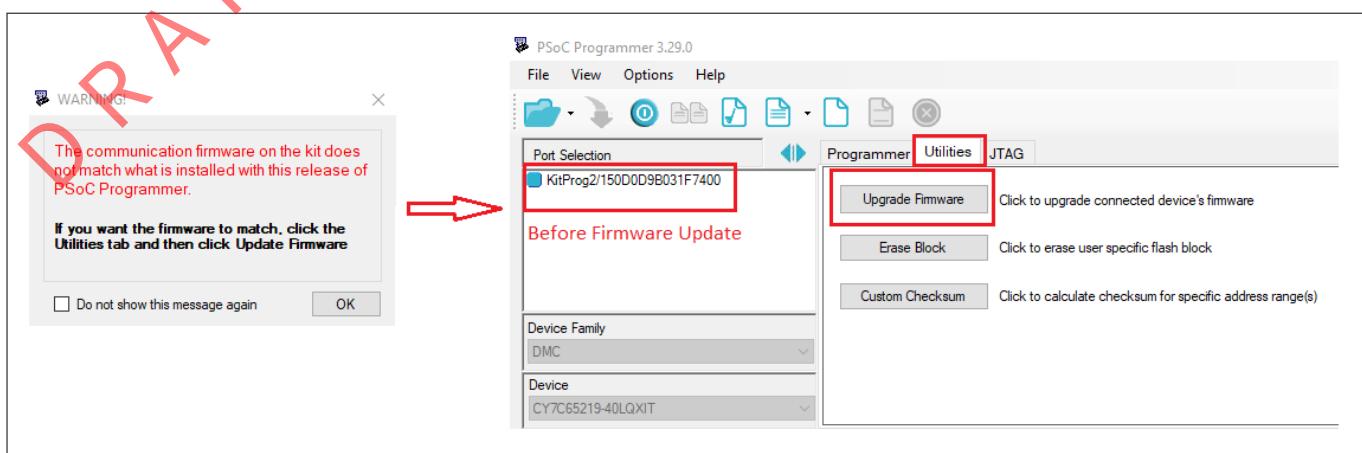


Figure 68 Updating KitProg firmware

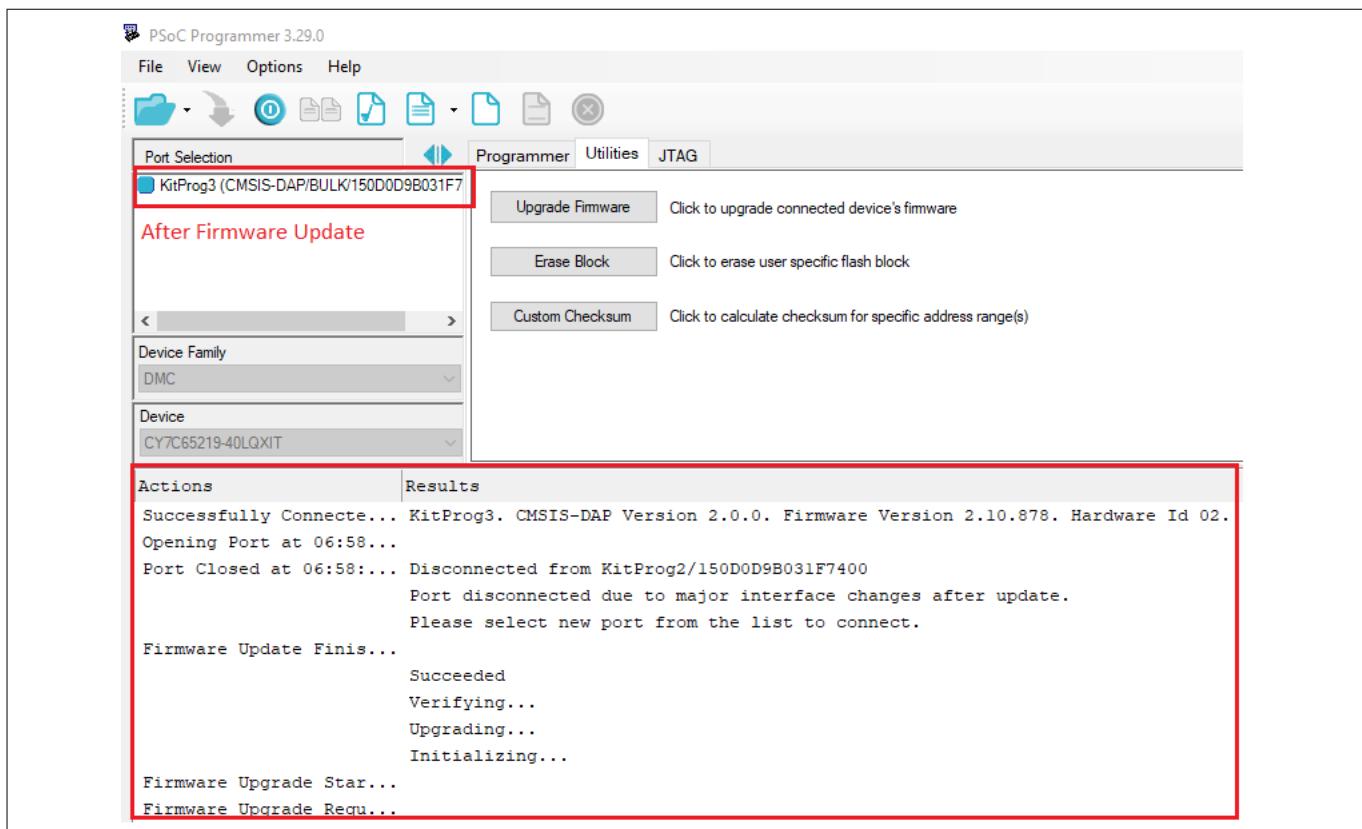


Figure 69 Firmware updates complete

5.3.2.6 CY8CKIT-062-BLE PSoC™ 6-BLE pioneer kit

The [Appendix D.1](#) is a Bluetooth® LE development kit from Infineon that supports the PSoC™ 63BL family of devices. See CY8CKIT-062-BLE PSoC™ 6-BLE pioneer kit in [Appendix D](#) for more information.

5.3.2.7 CySmart Host Emulation Tool and mobile applications

The CySmart Host Emulation Tool is a Windows application that emulates a Bluetooth® LE central device using the PSoC™ 6-BLE pioneer kit's Bluetooth® LE dongle. See [Appendix D.2](#) in [Appendix D](#) for more information. Similar functionality is available in the [Appendix D.3](#) for the iOS and Android devices. This tool is extremely useful in testing your Bluetooth® LE application.

~~DRAFT~~ 5 PSoC™ 6 application notes

5.3.3 Device features

The PSoC™ 6-BLE device has an extensive feature set as shown in [Figure 70](#). The following is a list of its major features. For more information, see the device [datasheet](#), the [Technical reference manual \(TRM\)](#), and the section on [Related application notes and code examples](#).

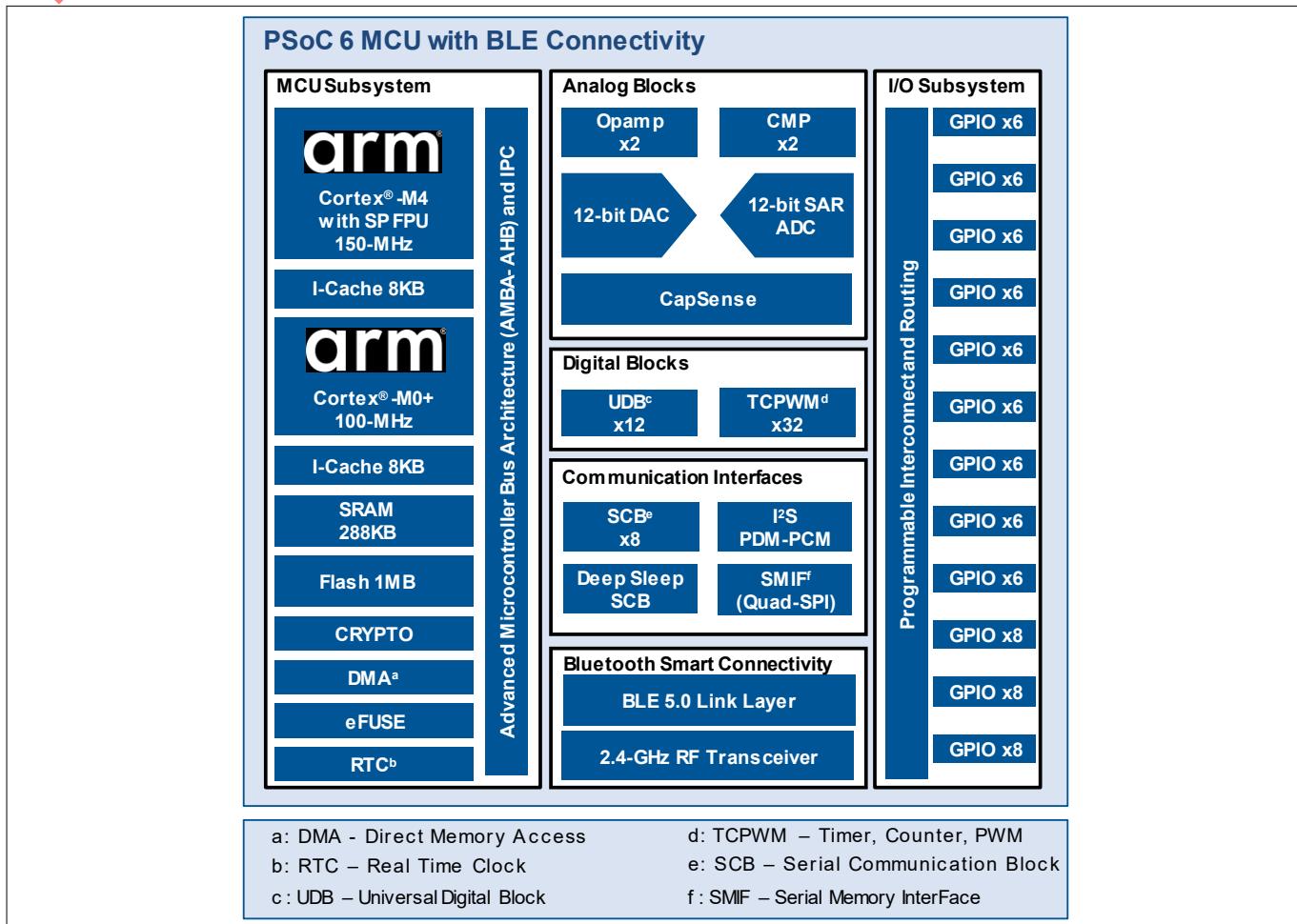


Figure 70 PSoC™ 6 MCU with Bluetooth® LE connectivity block diagram

- 32-bit dual-CPU MCU subsystem
 - 150-MHz CM4 CPU with single-precision floating-point unit 100-MHz CM0+ CPU
 - Up to 1 MB of user flash, with additional 32 KB for EEPROM emulation and 32 KB supervisory
 - Up to 288 KB of SRAM with selectable Deep Sleep retention granularity at 32-KB retention boundaries
 - Inter-processor communication supported in hardware
 - Cryptography accelerators with support for hardware acceleration and true random number generator function
 - Two 32-channel DMA controllers
 - eFUSE: one-time programmable bits
 - Uninterruptible secure boot with hardware hash-based authentication
- I/O subsystem
 - Up to 78 GPIOs that can be used for analog, digital, CAPSENSE™, or segment LCD functions
 - Programmable drive modes, drive strength, and slew rates
 - Two ports with smart I/Os that can implement boolean operations

5 PSoC™ 6 application notes

- DRAFT**
- Programmable analog blocks
 - Two opamps of 6-MHz gain bandwidth (GBW), configurable as programmable gain amplifiers (PGA), comparators, or filters
 - Two low-power comparators with less than 300 nA current consumption that are operational in Deep Sleep and Hibernate modes
 - One 12-bit, 1-MspS SAR ADC with 16-channel sequencer
 - One 12-bit voltage mode DAC
 - CAPSENSE™ with SmartSense™ auto-tuning
 - Supports CAPSENSE™ Sigma-Delta (CSD) and CAPSENSE™ transmit/receive (CSX)
 - Provides best-in-class SNR, liquid tolerance, and proximity sensing
 - Programmable digital blocks, Communication interfaces
 - 12 UDBs for custom digital peripherals
 - 32 TCPWM blocks configurable as 16-bit/32-bit timer, counter, PWM, or quadrature decoder
 - Nine SCBs configurable as I²C master or slave, SPI master or slave, or UART
 - Audio subsystem with one I²S interface and two PDM channels
 - SMIF interface with support for execute-in-place from external quad SPI flash memory and on-the-fly encryption and decryption
 - Bluetooth® connectivity with Bluetooth® Low Energy 4.2
 - 2.4-GHz Bluetooth® LE radio with integrated balun
 - Bluetooth® 4.2 specification-compliant controller and host implementation
 - Link layer engine that supports master/slave modes and up to four simultaneous connections
 - Support for 2-Mbps LE data rate
 - Operating voltage range, power domains, and low-power modes
 - Device operating voltage: 1.71 V to 3.6 V
 - User-selectable core logic operation at either 1.1 V or 0.9 V
 - Multiple on-chip regulators: low-drop out (LDO for Active, Deep Sleep modes), single input multiple output (SIMO) buck converter
 - Active, Low-Power Active, Sleep, Low-Power Sleep, Deep Sleep, and Hibernate modes for fine power management
 - Deep Sleep mode with operational Bluetooth® LE link: 4.5- μ A typical current at 3.3 V with 64-KB SRAM retention
 - An “always on” backup power domain with built-in RTC, power management integrated circuit (PMIC) control, and limited SRAM backup

~~5 PSoC™ 6 application notes~~

~~5.3.4 Development setup~~

~~DRAFT~~ Figure 71 shows the hardware and software tools required for evaluating Bluetooth® LE peripheral designs using the PSoC™ 6-BLE device. In a typical use case, the PSoC™ 6-BLE pioneer kit (blue board in Figure 71) is configured as a Peripheral that can communicate with a Central device such as the CySmart iOS/Android app or CySmart Host Emulation Tool. The CySmart Host Emulation Tool also requires a Bluetooth® LE dongle (black board in Figure 71) for its operation. The PSoC™ 6-BLE pioneer kit includes a dongle.

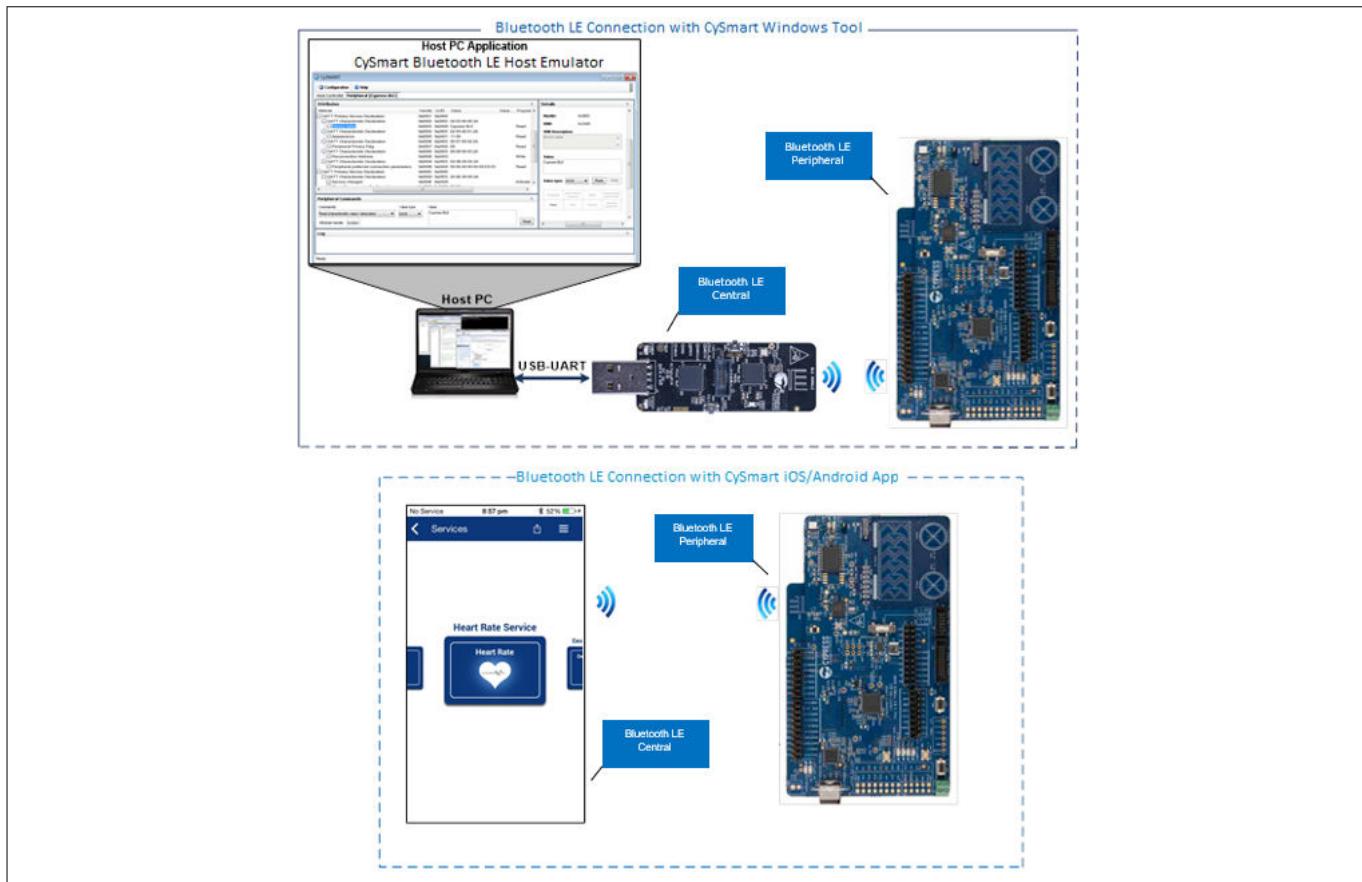


Figure 71 **Bluetooth LE functional setup**

As shown in Figure 72, the Bluetooth® LE dongle is preprogrammed to work with the Windows CySmart Host Emulation Tool. The PSoC™ 6-BLE pioneer kit has an onboard USB programmer that works with PSoC™ Creator for programming or debugging your Bluetooth® LE design. The Bluetooth® LE pioneer kit can be powered over the USB interface. Both the Bluetooth® LE dongle and the Bluetooth® LE pioneer kit can be connected simultaneously to a common host PC for development and testing.

5 PSoC™ 6 application notes

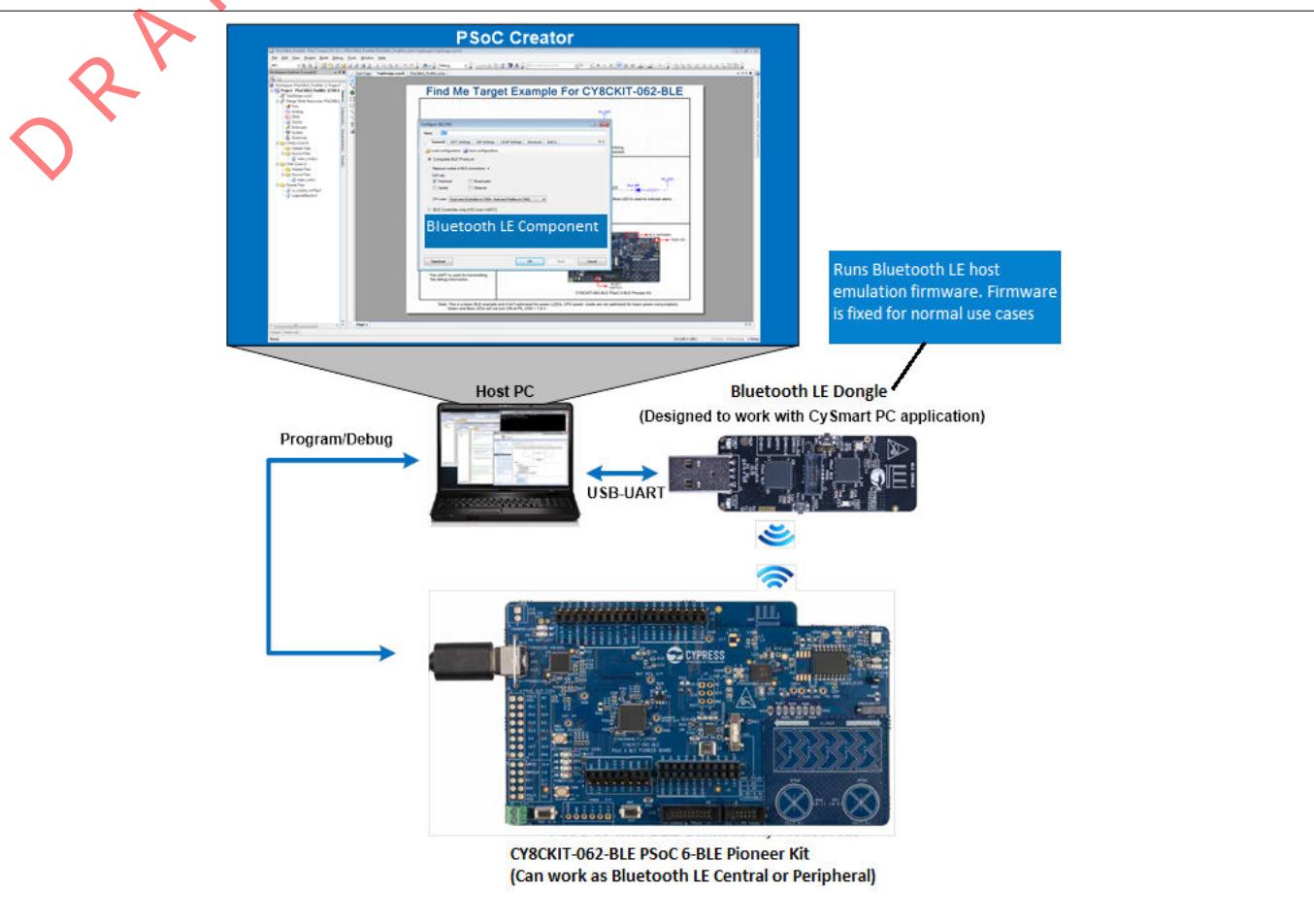


Figure 72 Bluetooth® LE development setup

~~DEAF~~ 5 PSoC™ 6 application notes

5.3.5 My first PSoC™ 6 MCU design with Bluetooth® LE

This section provides step-by-step instructions to build a simple design for the PSoC™ 6-BLE pioneer kit. A simple Bluetooth® SIG-defined standard profile called [Find Me Profile \(FMP\)](#) is implemented in the design. For creating advanced standard or custom Profile designs, refer to the Designing PSoC™ 6-BLE Applications and Creating a BLE Custom Profile application notes.

While the PSoC™ 6-BLE pioneer kit is intended to simplify and streamline design for Bluetooth® LE based applications, you can also use this setup to develop applications that do not use Bluetooth® LE.

5.3.5.1 Using the instructions

These instructions are grouped into several sections. Each section is devoted to a phase of the application development workflow. The major sections are:

- [Part 1. Create a new project from scratch](#)
- [Part 2. Implement the design](#)
- [Part 3. Generate source code](#)
- [Part 4. Write the firmware](#)
- [Part 5. Build the project, program the Device](#)
- [Part 6. Test your design](#)

These instructions require that you use a code example. However, the extent to which you use the code example depends on the path you choose to follow through these instructions.

We have defined three paths through these instructions:

Path	Working from scratch code example as reference Only	Using code example new to PSoC™ Creator or Bluetooth® LE	Using code example familiar with PSoC™ Creator and Bluetooth® LE
Best For	Someone who wants hands-on experience to learn about PSoC™ Creator and/or Bluetooth® LE	Someone who is new to the tool or technology, and wants to see how it all works	Someone who knows the tools and technology, and wants to see it work on the PSoC™ 6 platform

What you need to do for each path is clearly defined at the start of each part of the instructions.

If you start from scratch and follow all instructions in this application note, you use the code example as a reference while following the instructions. Working from scratch teaches you the most about the PSoC™ Creator design process, and how BLE works. This path also takes the most time.

You can also use the code example directly. It is a complete design, with all firmware written. You can walk through the instructions and observe how the steps are implemented in the code example. This is a particularly useful approach if you are already familiar with PSoC™ Creator and want to see how the Bluetooth® Low Energy Component is configured, for example. If you use the code example directly, you will be able to skip some steps.

In all cases, you should start by reading [Before you begin](#) and [About the design](#).

5.3.5.2 Before you begin

Ensure that you have the following items installed on your host computer.

- [PSoC™ Creator 4.4](#)
- [PDL v3.1.x](#)
- The [CySmart Host Emulation Tool](#) or the CySmart [iOS/Android](#) app

~~5 PSoC™ 6 application notes~~

- CE217236 – the code example used for this application note and
- The CY8CKIT-062-BLE PSoC™ 6-BLE pioneer kit with Kitprog3. Refer [Debugging](#) on how to update the Kitprog firmware

You can download the code example from the website by clicking the link above. You can also use the PSoC™ Creator **File > Code Example** command. Set the **Device family** to PSoC™ 63. Then set the **Filter by** option to **Find Me**. The CE212736_PSoC6BLE_FindMe code example appears. Select the code example, download by clicking on the download icon adjacent to the example. After installation is complete, click **Create Project**, and follow the on-screen instructions.

This design is developed for the [CY8CKIT-062-BLE PSoC™ 6-BLE pioneer kit](#). If you wish to use other hardware, you must adapt the instructions to your needs.

5.3.5.3 About the design

This design implements a Bluetooth® LE Find Me Profile ([FMP](#)) in the Target role that consists of an Immediate Alert Service ([IAS](#)). FMP and IAS are a Bluetooth® LE standard profile and service defined by the Bluetooth® SIG. Alert levels triggered by the Find Me Locator are indicated by varying the state of an LED on the Bluetooth® LE Pioneer Kit, as [Figure 73](#) shows. In addition, two status LEDs indicate the state of the Bluetooth® LE interface. There is also an optional UART component to print debug messages to a terminal window.

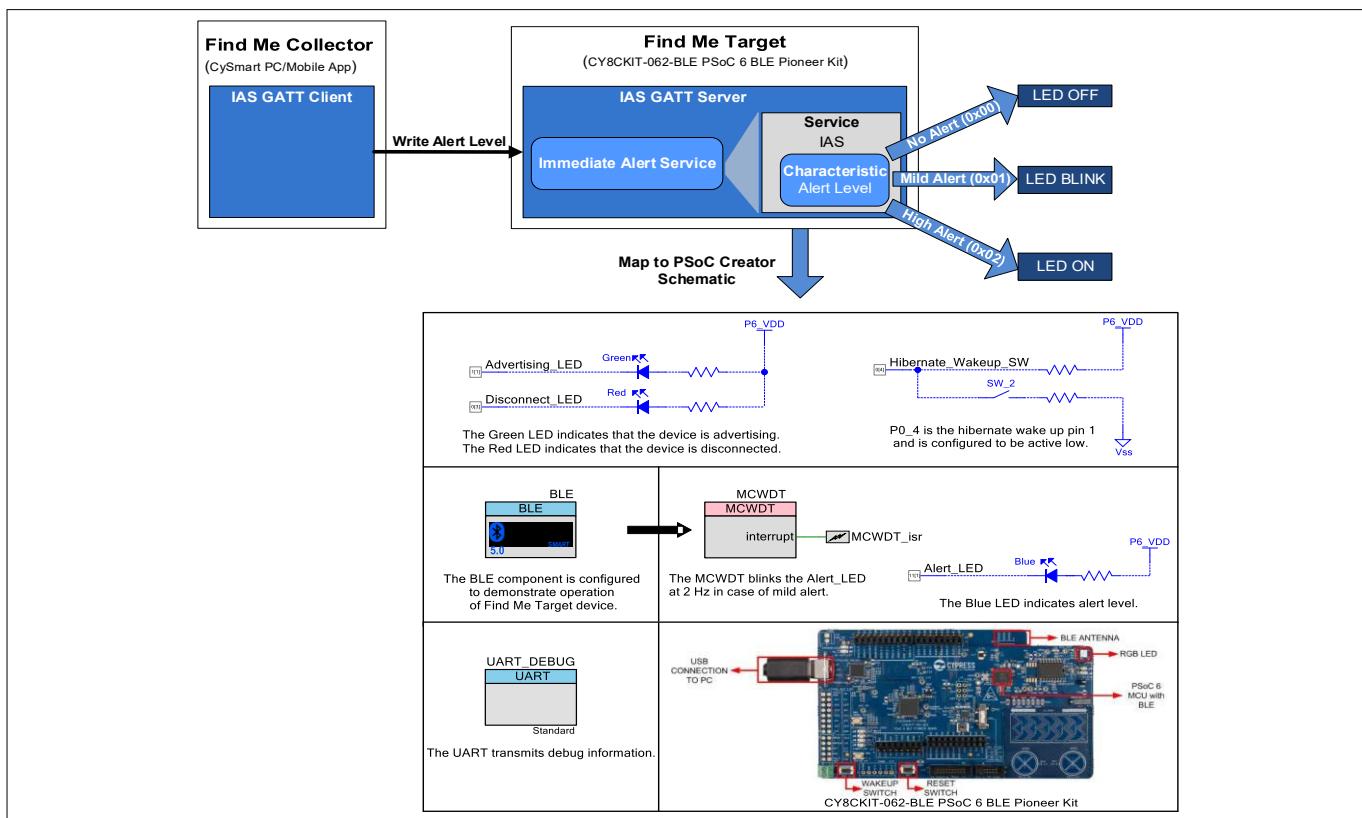


Figure 73 My first PSoC™ 6-BLE design

5.3.5.4 Part 1. Create a new project from scratch

This part takes you step-by-step through the process of creating a project file from scratch.

5 PSoC™ 6 application notes

DRAFT

Path	Working from scratch code example as reference only	Using code example new to PSoC™ Creator or Bluetooth® LE	Using code example familiar with PSoC™ Creator and Bluetooth® LE
Actions	Perform all steps	Do Step 1 Read the rest of the steps	Do Step 1, then jump to Part 2

Note: These instructions assume that you are using PSoC™ Creator 4.4 or higher. The overall development process is the same for subsequent versions of PSoC™ Creator; but the user interface may change over time.

Launch PSoC™ Creator and get started.

- Ensure that PSoC™ Creator can find the PDL

Attention: This should be set correctly automatically during installation, but nothing works if this isn't set up right. See [Figure 74](#) for help with this step.

- Choose menu that is **Tools > Options**
- On the Project Management panel, check the path in the PDL v3 (PSoC™ 6 devices) location field
- Ensure that it is correct. If it is not, click **Browse** and locate the installed directory of the PDL. The default location is C:\Program Files (x86)\Cypress\PDL\3.0.1

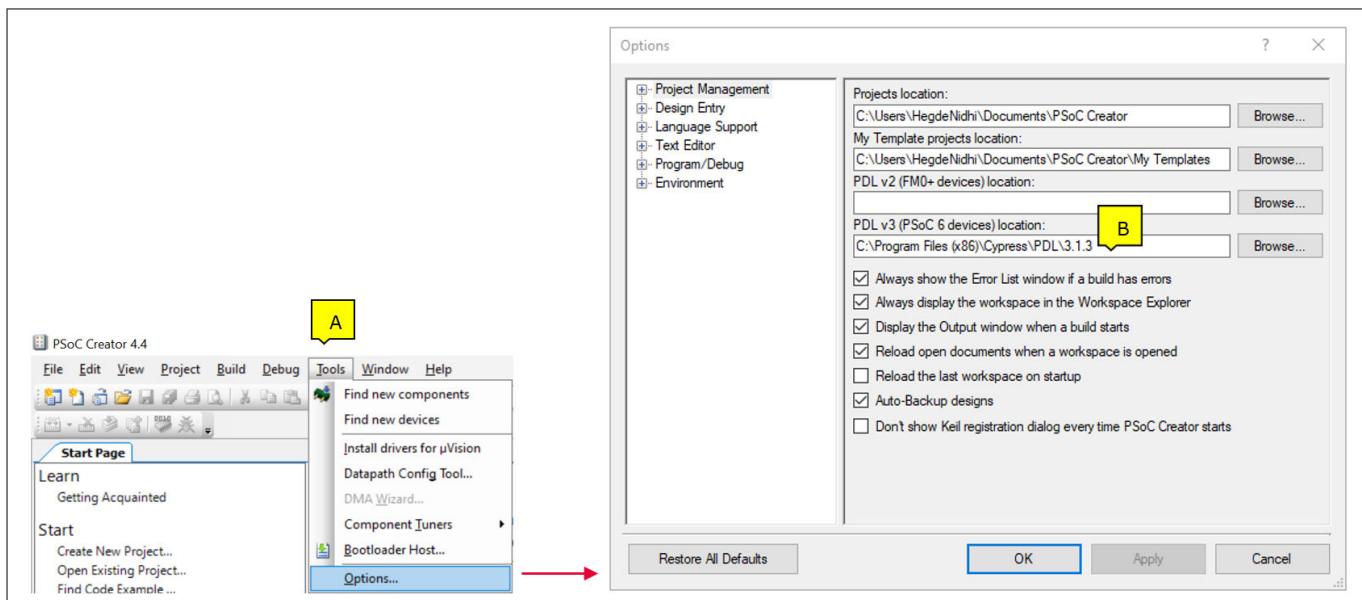


Figure 74 Peripheral Driver Library (PDL) location

Optional: Jump to [Part 2. Implement the design](#)

- Create a new PSoC™ Creator project

- Choose **File > New > Project**, as [Figure 75](#) shows. The **Create Project** window appears

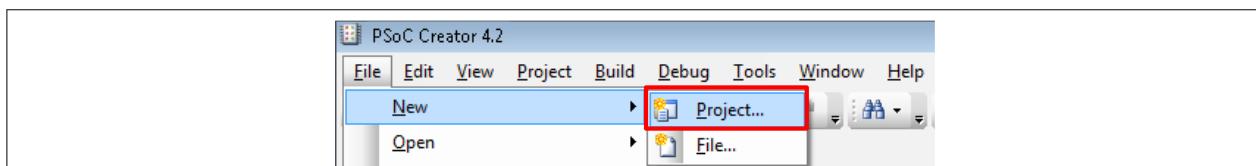


Figure 75 Create a new PSoC™ Creator project

5 PSoC™ 6 application notes

DRAFT

Note: If you are using the code example choose **File > Open > Project/Workspace**, as Figure 76 shows. The **Open** window appears. Point to the location of the Code Example workspace and open the workspace.

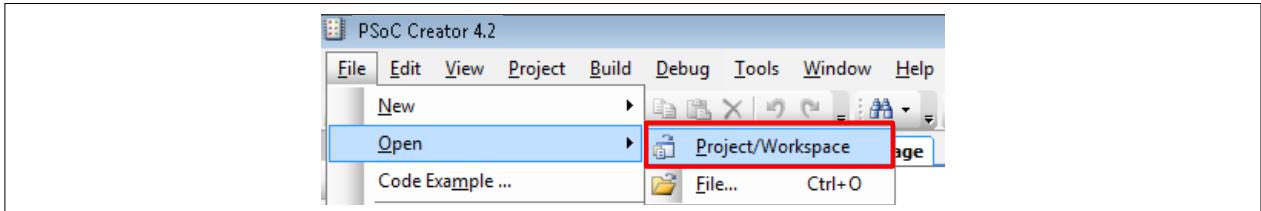


Figure 76 Open existing code example workspace

- **Select PSoC™ 6 Bluetooth® LE as the target device**

Tip: PSoC™ Creator speeds up the development process by automatically setting various project options for specified development kits or target devices. See [Figure 75](#) for help with this step.

- Select Target device
- From the family drop-down list, select PSoC™ 6
- From the device drop-down menu, list PSoC™ 63
- Click **Next**. The Select project template panel appears

PSoC™ Creator uses CY8C6347BZI-BLD53 as the default device in the PSoC™ 6 MCU with Bluetooth® LE family. This device is mounted on the CY8CKIT-062-BLE PSoC™-BLE Connectivity Pioneer Kit.

If you are intending to use custom hardware based on PSoC™ 6-BLE, or a different PSoC™ 6-BLE part number, this is the place you choose to **Launch device selector** option in **Target device** and select the appropriate part number.

5 PSoC™ 6 application notes

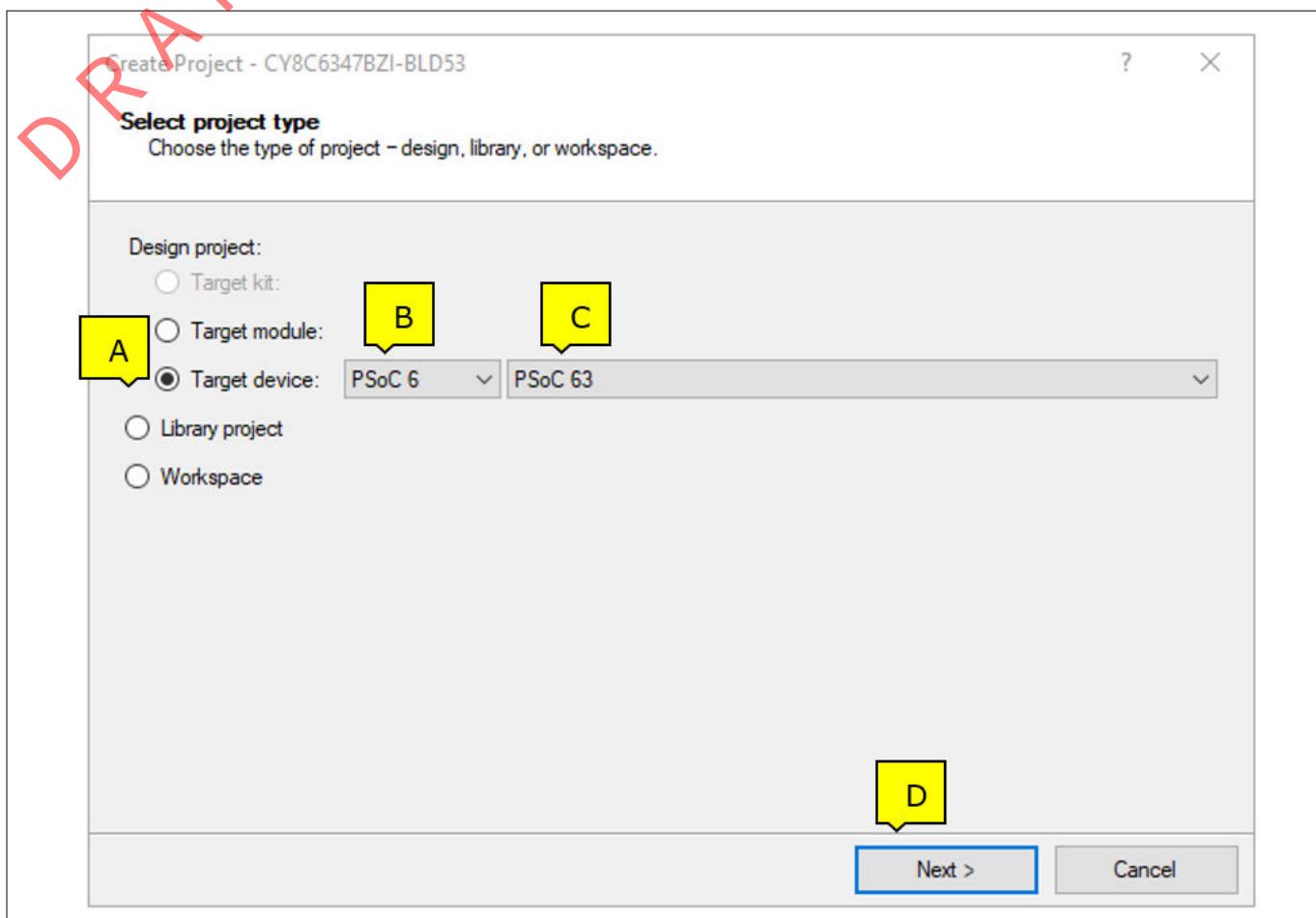


Figure 77 Selecting target device

- **Pick a project template**
 - Choose empty schematic
 - Click **Next**

5 PSoC™ 6 application notes

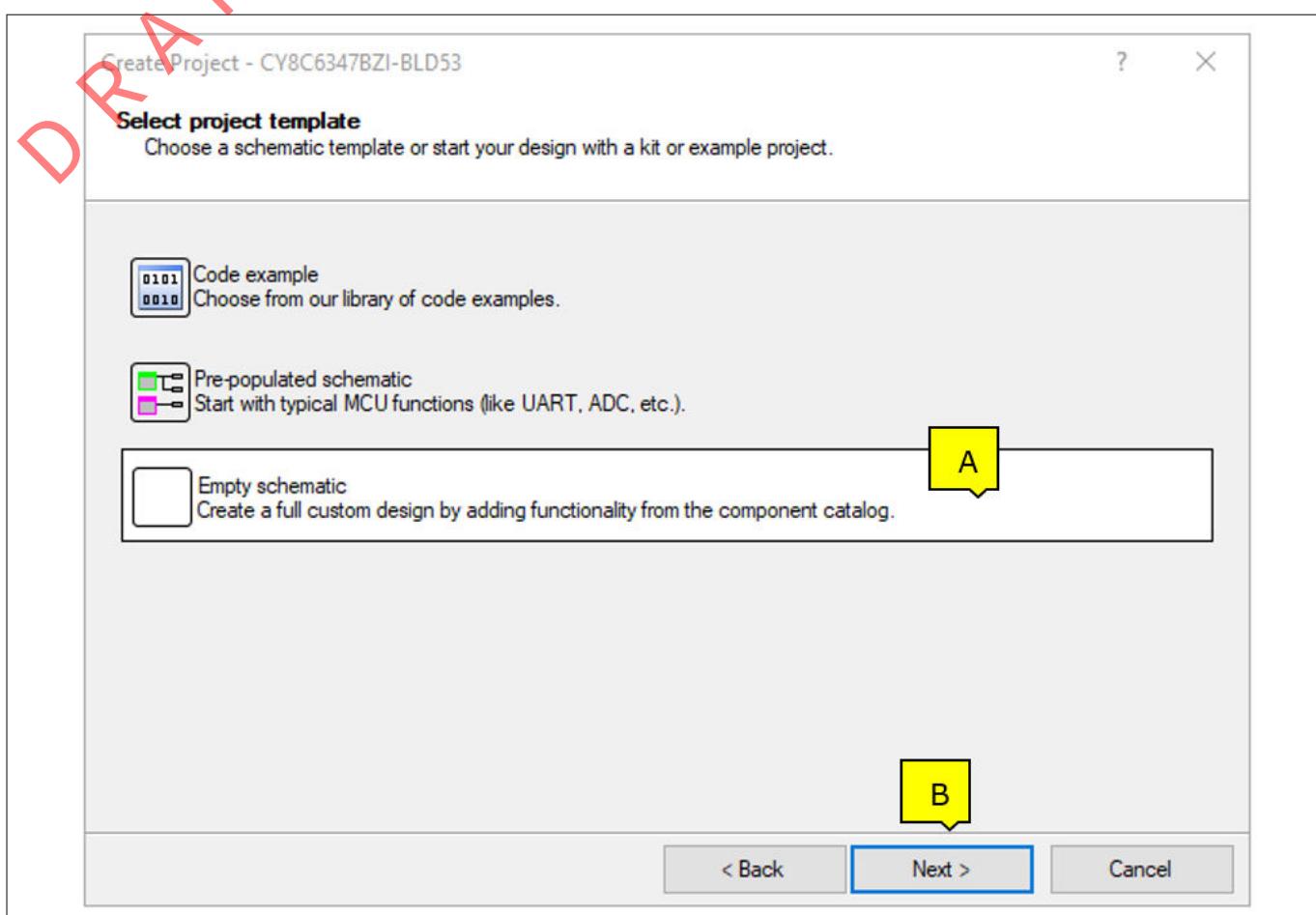


Figure 78 Pick a project template

- **Select target IDE(s)**

Note: If you expect to export the code from the project, specify the target IDE. By default, all export options are disabled. You can modify this setting later if circumstances change.

- Click **Next** to accept the default options

5 PSoC™ 6 application notes

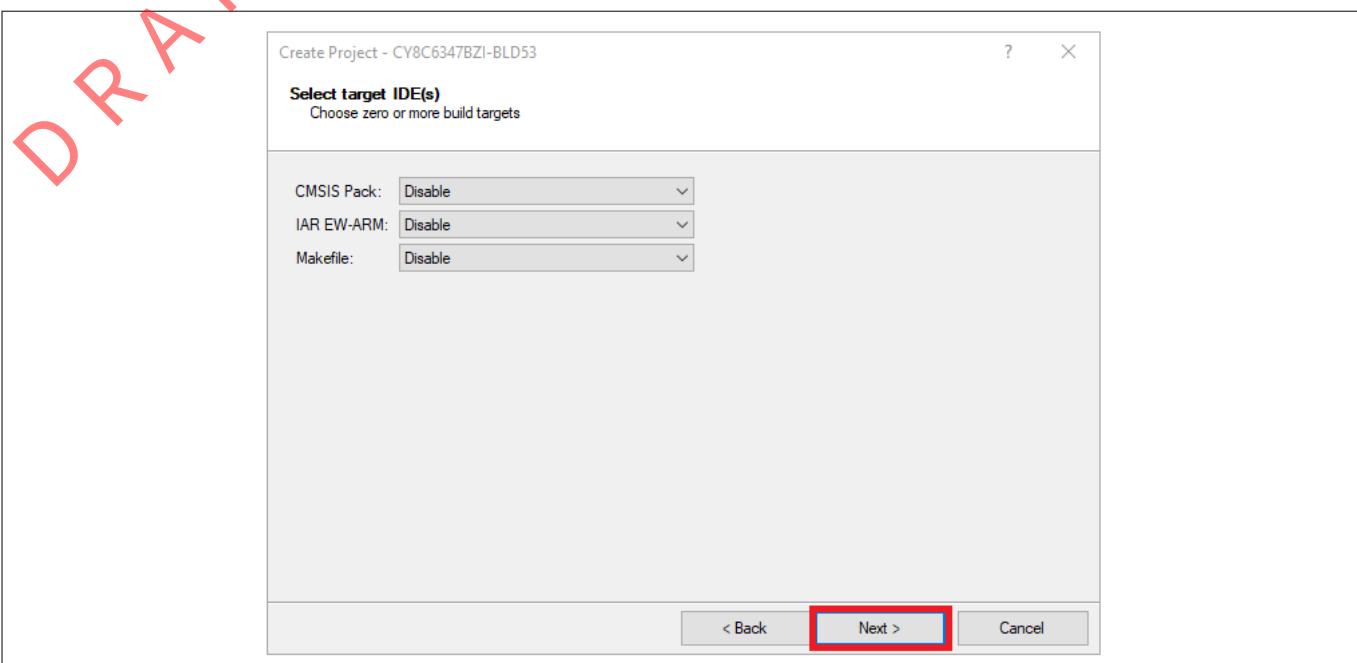


Figure 79 Select target IDEs (all disabled)

- **Create the project**

Note: In this step, you set the name and location for your workplace, and a name for the project. See [Figure 80](#) for help with this step. A workspace is a container for one or more projects.

- Set the Workspace name
- Specify the Location of your workspace
- Set a Project name. The project and workspace names can be the same or different
- Click **Finish**

5 PSoC™ 6 application notes

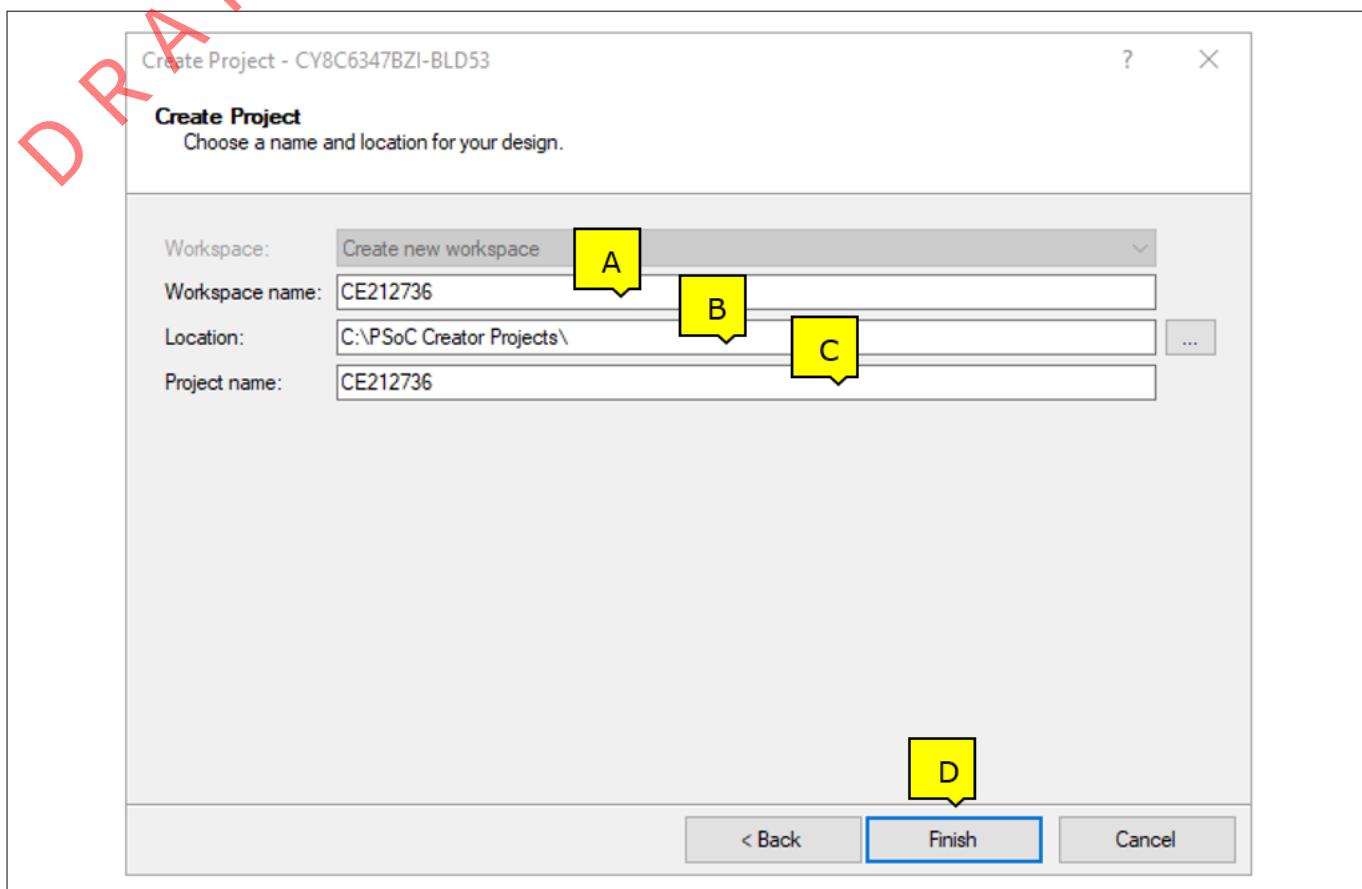


Figure 80 Project naming and location

You have successfully created a new PSoC™ Creator project.

5.3.5.5 Part 2. Implement the design

Now that you have a project file, it is time to implement the hardware design using PSoC™ Creator components. If you are using the code example directly, you already have a complete design. Perform the actions recommended based on your chosen path through this exercise.

Path	Working from scratch code example as reference only	Using code example new to PSoC™ Creator or Bluetooth® LE	Using code example familiar with PSoC™ Creator and Bluetooth® LE
Actions	Perform all steps	Read and understand all steps	You can skip this part if you wish. Jump to Part 3. Generate source code

Before you implement the design, a quick tour of the PSoC™ Creator interface is in order.

[Figure 81](#) shows the PSoC™ Creator application displaying an empty design schematic.

The project includes a project folder with a base set of files. You view these files in the **Workspace Explorer** pane to the left. The project schematic opens by default. This is the *TopDesign.cysch* file. Double-click the file name in the explorer pane to open the schematic at any time. In a new project, the schematic is empty. If you are using the code example, this is the schematic for the Find Me Bluetooth® LE application.

The Component catalog is on the right side of the window. You can open it with the **View > Component Catalog** menu item. You can search for a Component by typing the name of the Component in the **Search for...** text box and then pressing the enter key. See [Figure 81](#).

5 PSoC™ 6 application notes

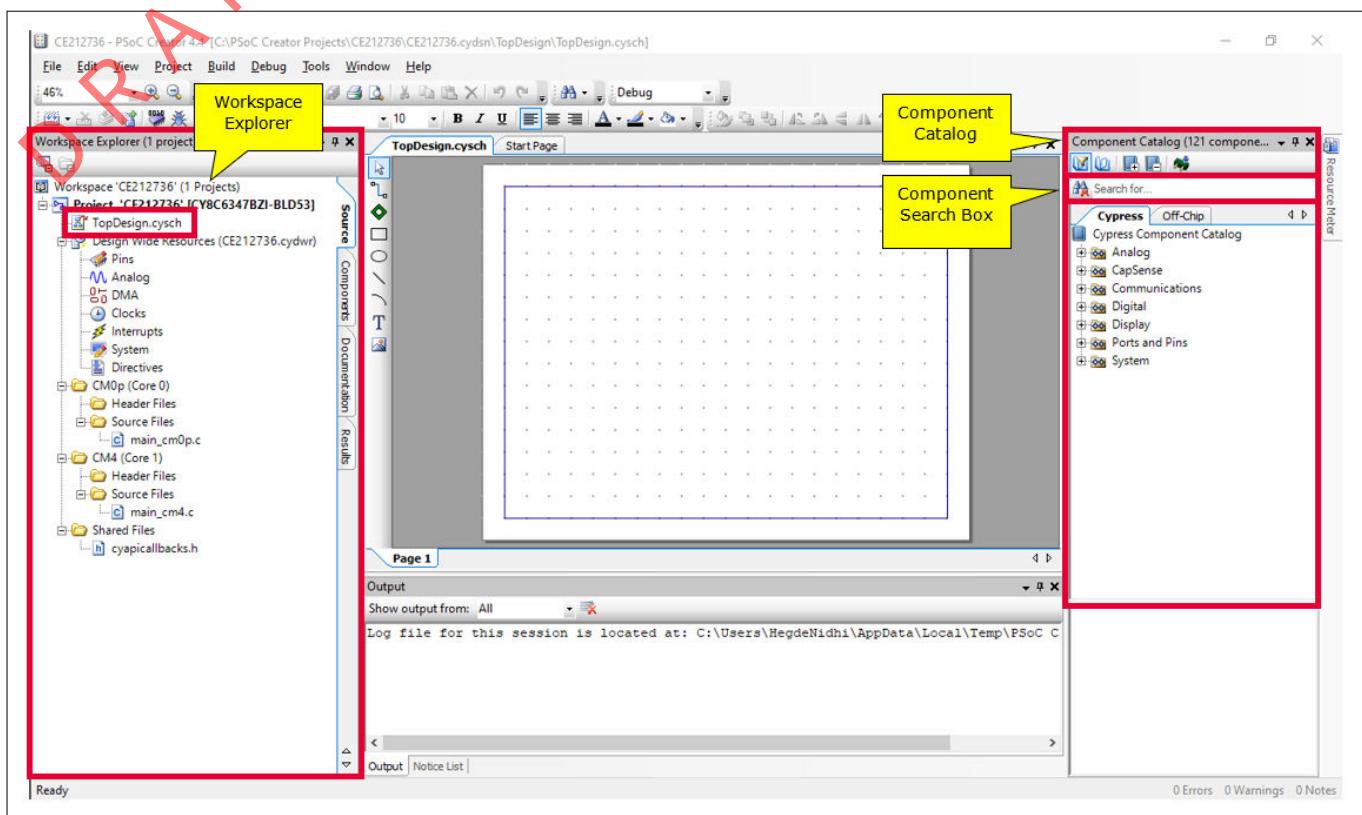


Figure 81 Schematic and component catalog

- **Place components in the design**

This design uses several components: the Bluetooth® Low Energy component, a digital input pin, three digital output pins, a UART, a Watchdog Timer, and an Interrupt. In this step, you add them to the design. You configure them in subsequent steps. [Figure 82](#) shows the result.

- In the **Component Catalog**, expand the **Communications** group, drag a **Bluetooth® Low Energy** component into the schematic, and drop it. It doesn't matter where you put a Component. Alternatively, you can search the Bluetooth® Low Energy Component by typing BLE in the Component Catalog search box
- Also in the **Communications** group, expand UART and drag a **UART (SCB)** Component into the design
- Expand the **Ports and Pins** group, drag a **Digital Input Pin** into the design
- From the same **Ports and Pins** group, drag a **Digital Output Pin** into the design. Repeat this twice, for a total of three pins
- Expand the **System** group, drag an **Interrupt** Component and a **MCWDT** Component into the design

5 PSoC™ 6 application notes

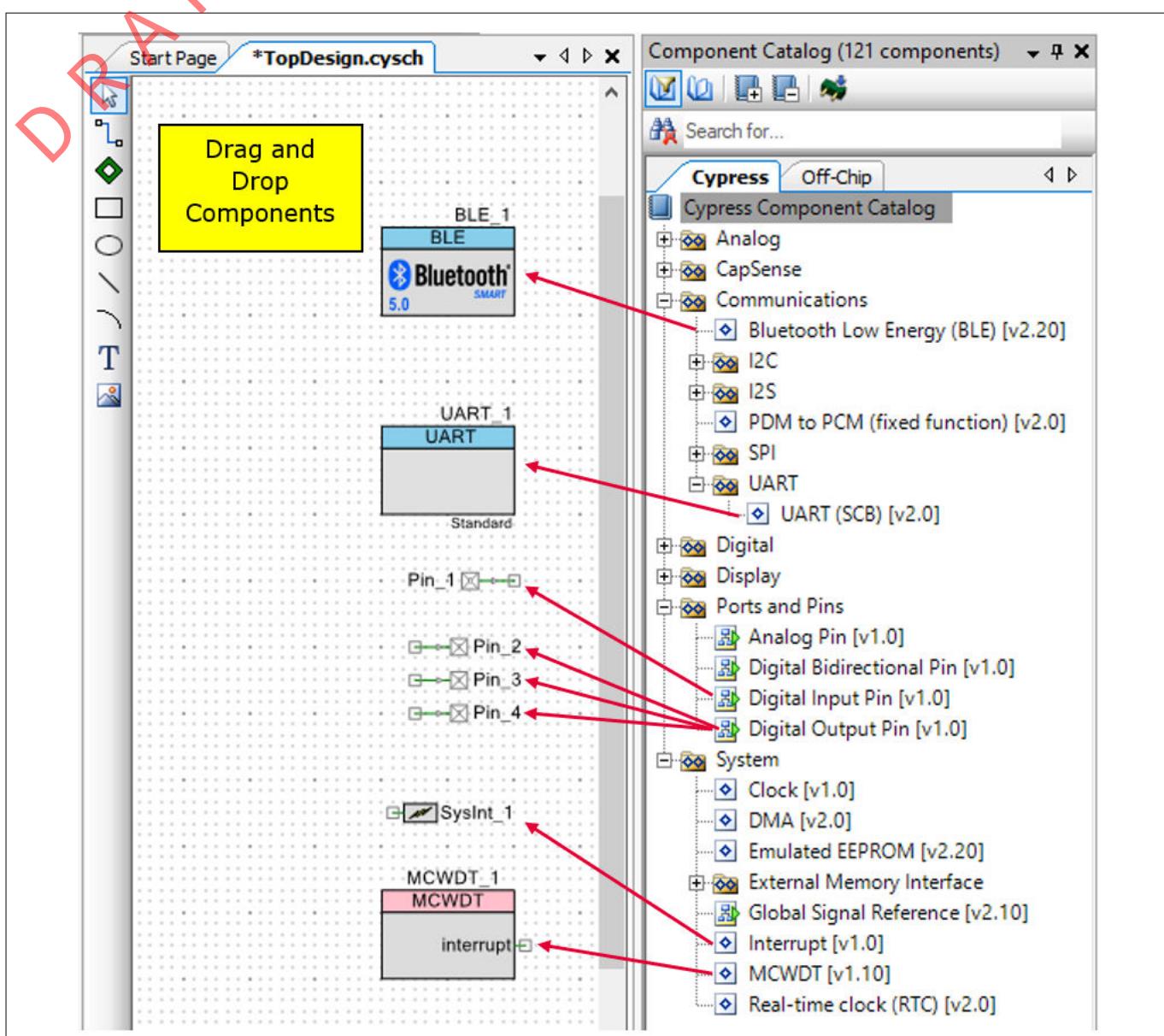


Figure 82 Place components in the design

PSoC™ Creator gives each Component a default name and properties. Default values may or may not be suitable for any given design. In subsequent steps you modify the name and some of the properties.

- Configure the three LED pins

One pin drives the alert status LED. The other two drive the Bluetooth® LE advertising and disconnection indicator LEDs. An LED on the Bluetooth® LE pioneer kit is active LOW; that is, the logic high pin-drive state turns OFF the LED, and the logic low pin-drive state turns it ON.

All three pins are configured identically, except for the name. Repeat these instructions three times, once for each pin. [Figure 83](#) shows the configuration.

Double-click the component placed on the schematic to open the configuration dialog. Then perform the following steps.

- Change the name of the Component instance for each pin. The three names are **Alert_LED**, **Advertising_LED** and **Disconnect_LED**
- For each pin, deselect **HW connection**. The firmware will drive the pin
- For each pin, set **Initial drive state** to **High (1)**. Hence, by default the LEDs will be OFF

5 PSoC™ 6 application notes

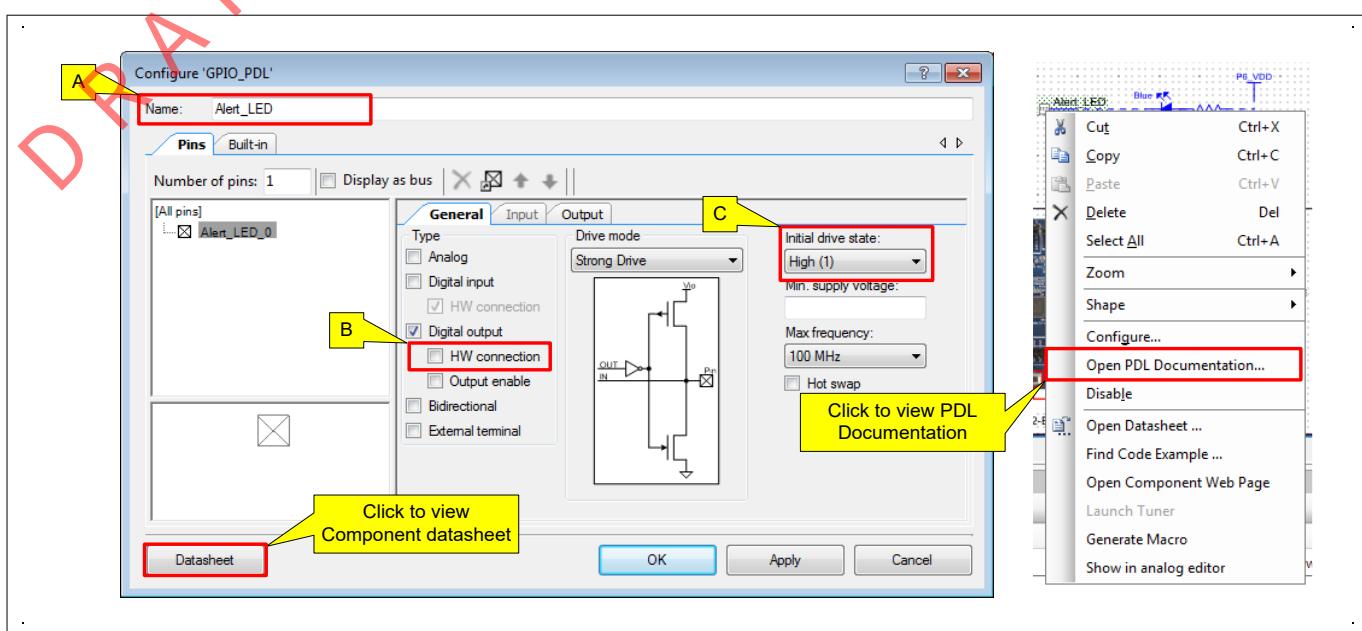


Figure 83 Configuring an output pin component

Make sure you configure all three pins.

Tip: Each component has an associated datasheet that can be accessed from the configuration window. The Component datasheet provides more information on the Component configuration, the application programming interface (API), and the electrical specifications.

Tip: You can open the API reference document of associated PDL driver of a Component by right-clicking the component and clicking the **Open PDL Documentation...** link. See [Figure 83](#).

Tip: For a pin, if you enable **External terminal** you can add external “off-chip” Components to a design. External Components on the schematic are included for descriptive purposes only; they have no effect on the generated code. Off-chip Components are optional, but can assist the hardware design team understanding how the design works. You can also add text boxes to a design with descriptions.

[Figure 84](#) shows how you could enhance the design for the Alert LED. In this case, the off-chip components were configured with **Instance_Name_Visible** unchecked. The resistors were configured with the **Value** field left blank. The power terminal was configured with the **Supply_Name** set to **P6_VDD**.

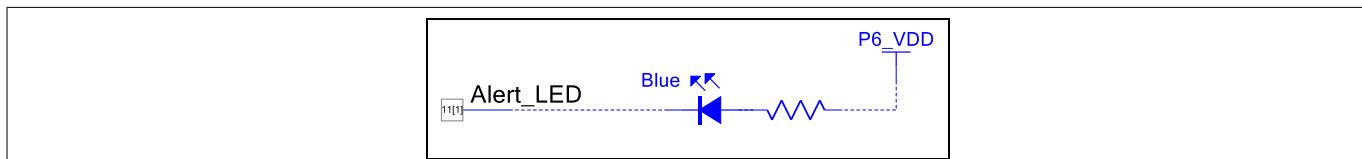


Figure 84 An output pin with off-chip components

- **Configure the Hibernate wakeup pin**

Switch SW2 on the Pioneer kit is connected to one of the Hibernate wakeup pins, P0[4] and when pressed pulls the port pin LOW. To configure this pin as the Hibernate wakeup switch, it must be configured as resistive pull up.

[Figure 85](#) shows the configuration. Double-click the component placed on the schematic to open the configuration dialog, and then do the following:

5 PSoC™ 6 application notes

- Change the name of the Component instance to **Hibernate_Wakeup_SW**
- Deselect **HW connection**. The firmware application firmware does not drive this pin. However, the pin is hard-wired to the Hibernate system. In the firmware, wakeup will be configured as active LOW
- Set **Drive mode** to **Resistive Pull Up** and **Initial drive state** to **High (1)**. This will configure the device to detect the HIGH to LOW transition and wake up the device from Hibernate

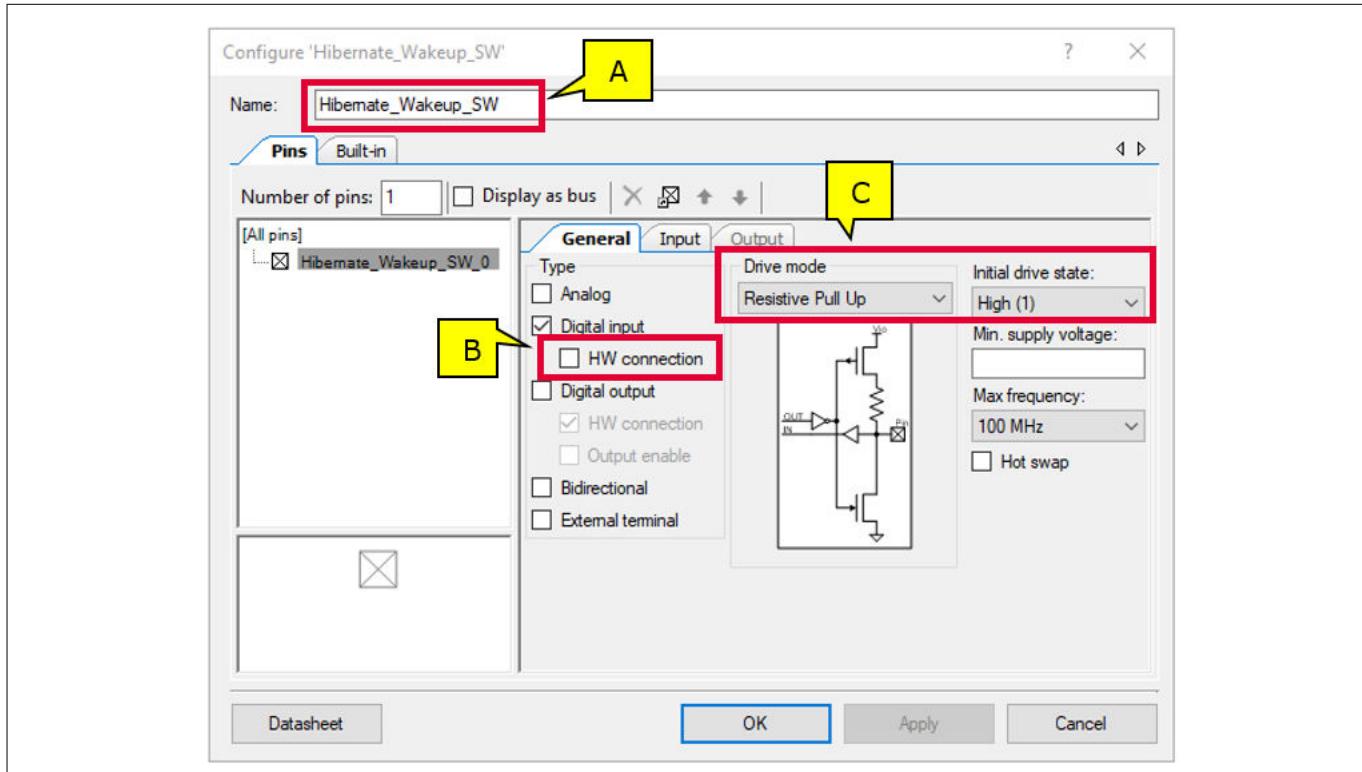


Figure 85 Configuring an input pin component

- Configure the **UART** component

Double-click the component placed on the schematic to open the configuration window. The design uses this Component to display debug messages in a terminal window at a baud rate of 115200 bps. It is not related to Bluetooth® LE functionality.

- Change the **Name** of the Component instance to “**UART_DEBUG**”
- Click **OK**

The design uses default values for all other settings.

5 PSoC™ 6 application notes

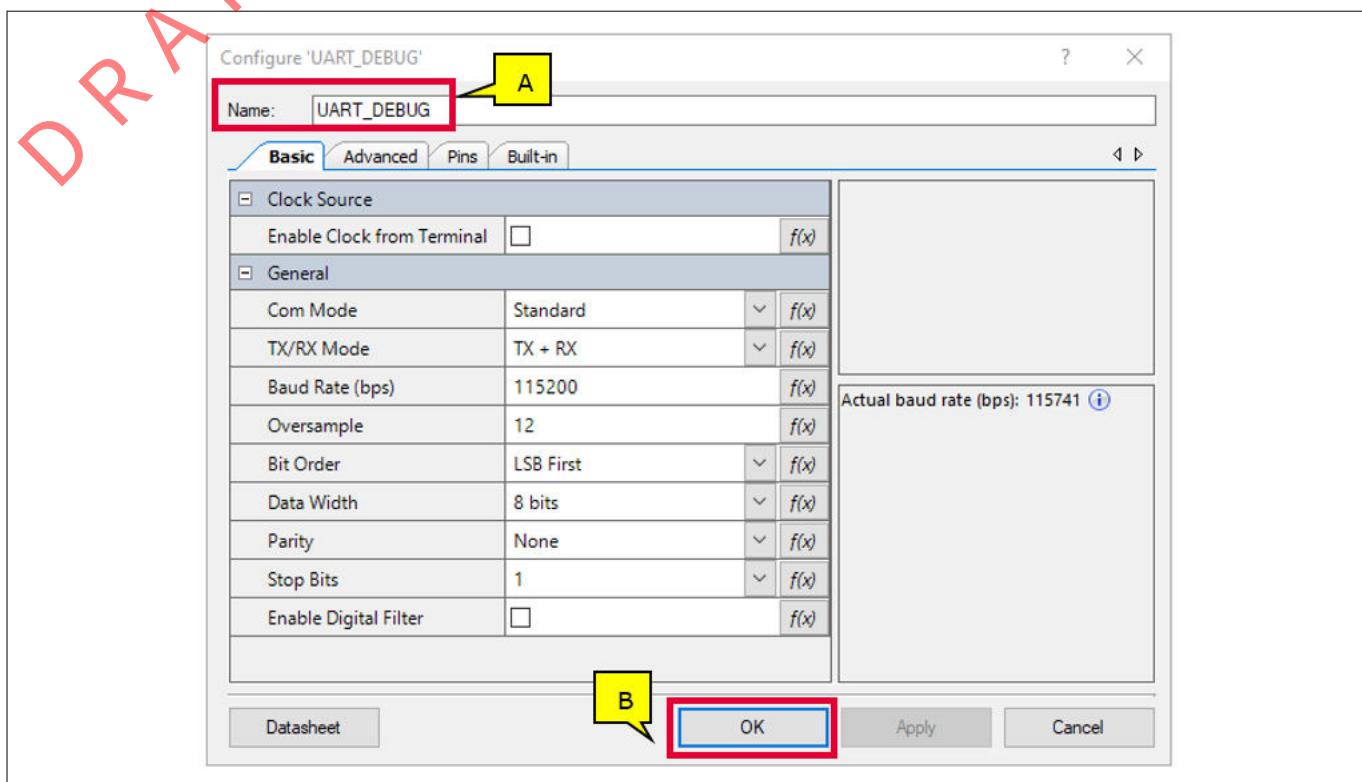


Figure 86 Configuring the SCB-based UART component

- **Set the general Bluetooth® LE options**

Double-click the Bluetooth® Low Energy Component placed on the schematic to open the configuration window. Set the **General** properties as shown in [Figure 87](#). Except for the Component name and the stack operation, this application uses default general properties.

- Change the **Name** to “BLE”
- Confirm that **Complete BLE Protocol** is selected
- Change **Maximum number of BLE connections** to 1. This will configure the BLE stack appropriately
- Confirm that **Peripheral** is selected as the **GAP role**. This sets the device to act as a BLE Peripheral device and respond to Central device requests
- Select **Dual core (Controller on CM0+, Host and Profiles on CM4)** option for **CPU core**. This will split the Bluetooth® LE stack to work on both the cores. The CM0+ core runs the BLE controller portion of the stack and is responsible for maintaining the BLE connection. The BLE Host runs on the CM4 core and performs application-level tasks. The main advantage of this dual-CPU setup is that the CM4 core can go into Deep Sleep low-power mode when there are no Bluetooth® LE-related tasks pending

5 PSoC™ 6 application notes

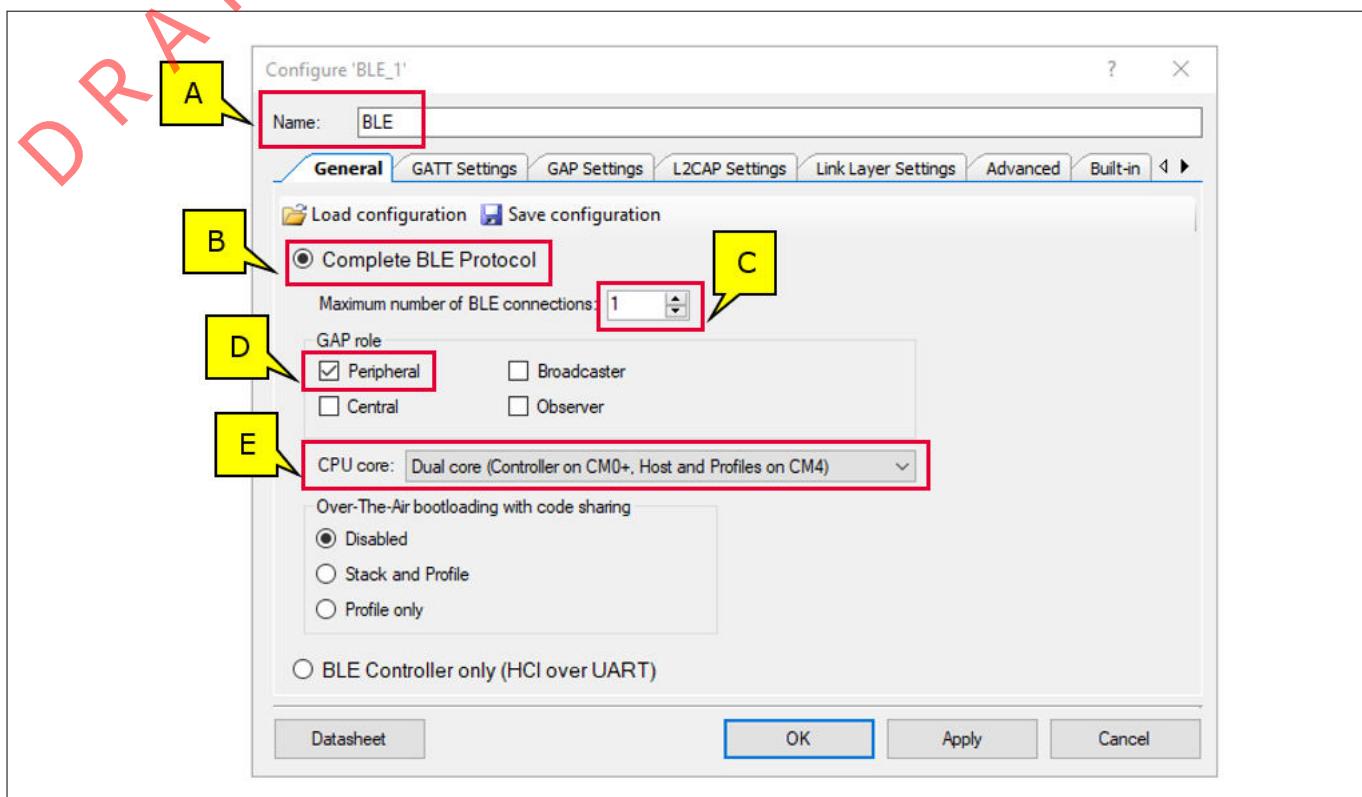


Figure 87 Bluetooth® Low Energy component general configuration

- Specify the Generic Attribute (GATT) settings

In this step, you set the Bluetooth® LE profile as shown in [Figure 88](#).

- Click the **GATT Settings** tab to display GATT options
- Click the **Add Profile** drop-down menu
- Select the **Find Me > Find Me Target (GATT Server)** option. This sets the GAP Peripheral role profile. When the menu disappears, notice that a new service “**Immediate Alert**” appears, as shown in [Figure 89](#)

5 PSoC™ 6 application notes

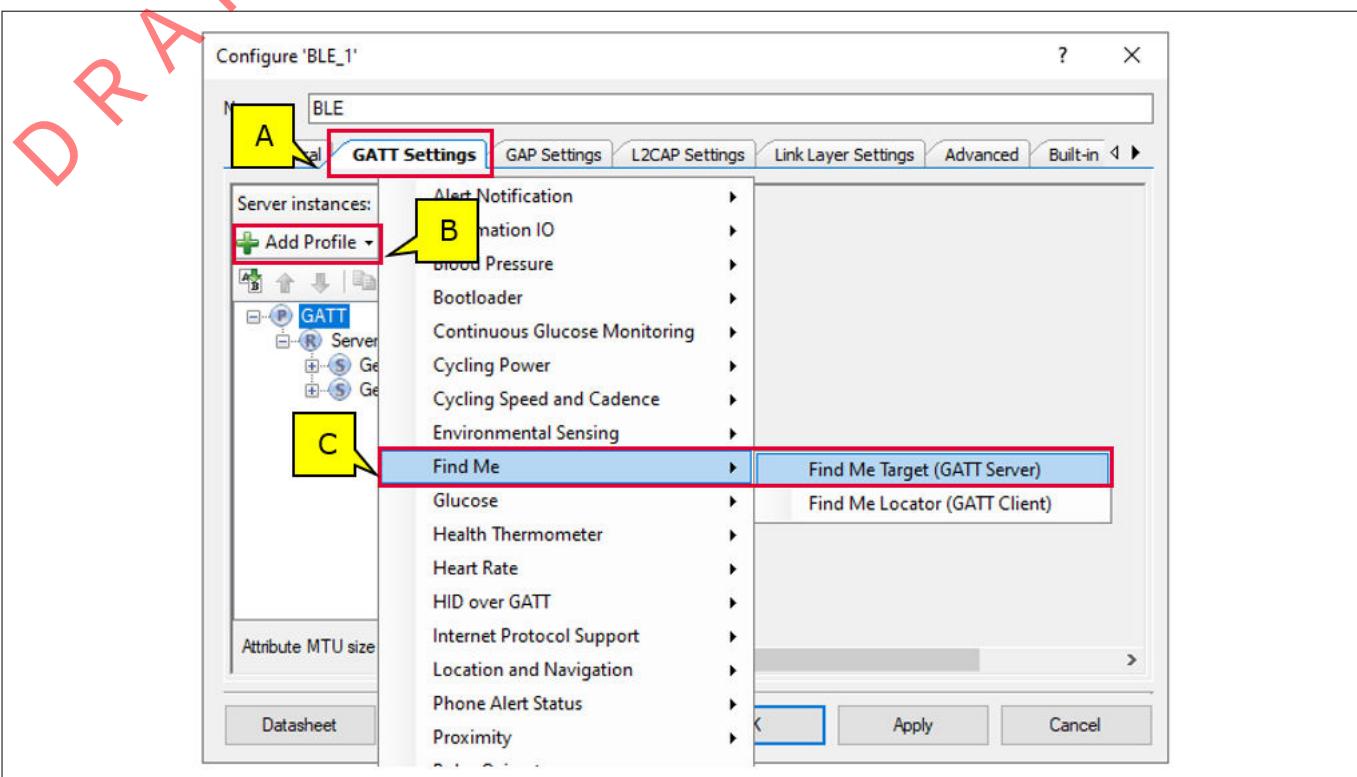


Figure 88 Bluetooth® Low Energy component immediate alert service added

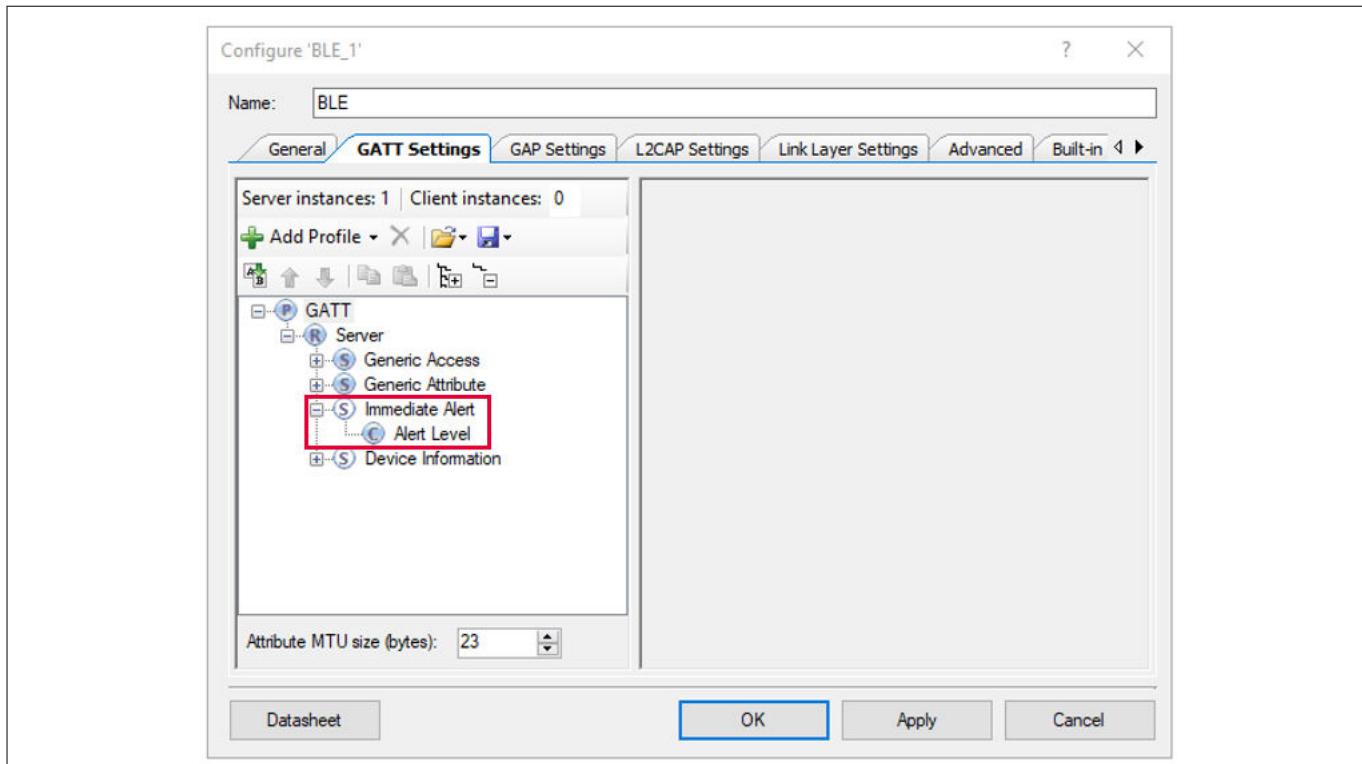


Figure 89 Bluetooth® Low Energy component immediate alert service added

Note: The code example has a **Device Information Service** also added in the GATT Settings to identify the device and read the firmware version. This service is not needed for **Immediate Alert** to work.

- Specify the Generic Access Profile (GAP) general settings

5 PSoC™ 6 application notes

There is a series of panels to cover GAP settings. The left menu provides access to all the panels. See [Figure 90](#).

- Click the **GAP Settings** tab to display GAP options. The **General** panel appears by default
- Enter **Find Me Target** as the **Device name**
- Set the **Appearance** to **Generic Keyring**

All other general settings use default values. This includes that the device uses **Silicon generated “Company assigned” part of device address**. This configures the device name and type that appears when a Host device attempts to discover your device, and then assigns a unique Bluetooth® LE device address to your device.

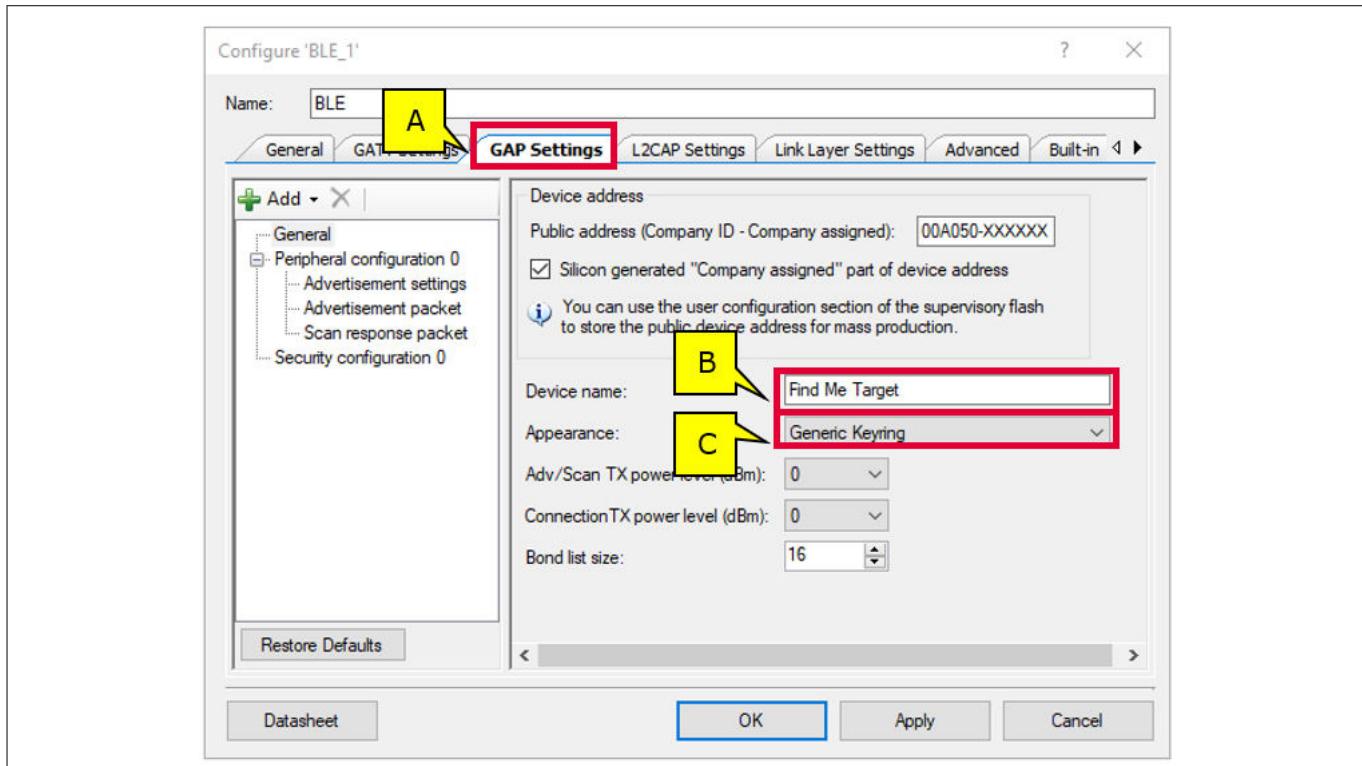


Figure 90 **Bluetooth® Low Energy component general GAP settings**

- Specify the GAP advertisement settings**

See [Figure 91](#) for help with this step.

- Click the **Advertisement settings** item in the left menu. The panel appears
- Set **Advertising type** to **Connectable undirected advertising**
- Deselect the **Slow advertising interval** checkbox

Other than that, default values work for this application. It uses limited discovery mode with an advertising timeout of 30 seconds and a fast advertisement interval of 20 to 30 ms. Fast advertising allows quick discovery and connection but consumes more power due to increased RF advertisement packets.

5 PSoC™ 6 application notes

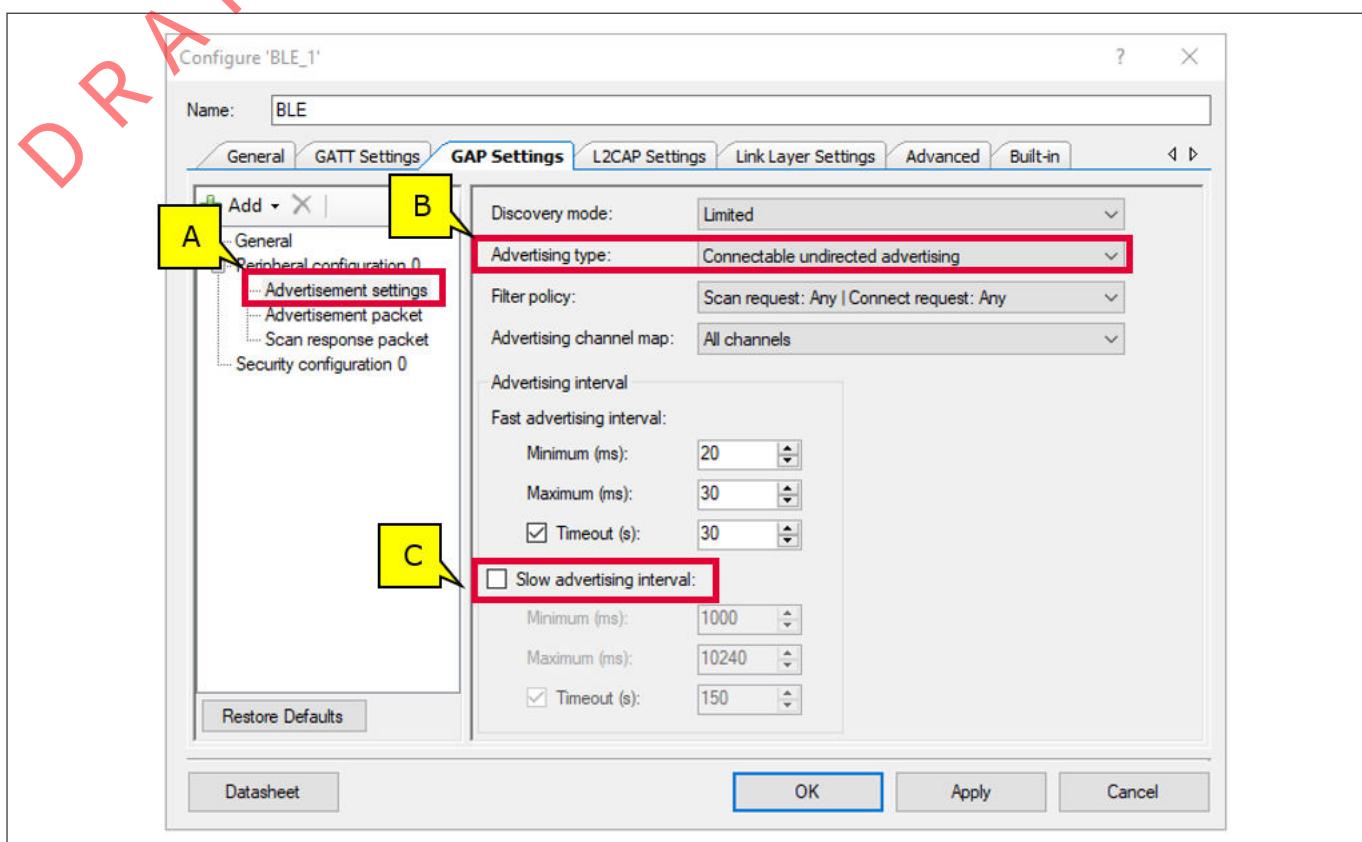


Figure 91 Bluetooth® Low Energy component GAP advertisement settings

- Specify the GAP advertisement packet settings

In this step, you enable the device for the Immediate Alert Service (IAS). See [Figure 92](#).

- Click the **Advertisement packet** item in the left menu. The panel appears
- Expand the **Service UUID** item, and select **Immediate Alert**

This configures the device to notify Bluetooth® Low Energy central devices that the IAS is available. As you add items, the structure and content of the advertisement packet appears to the right of the configuration panel.

5 PSoC™ 6 application notes

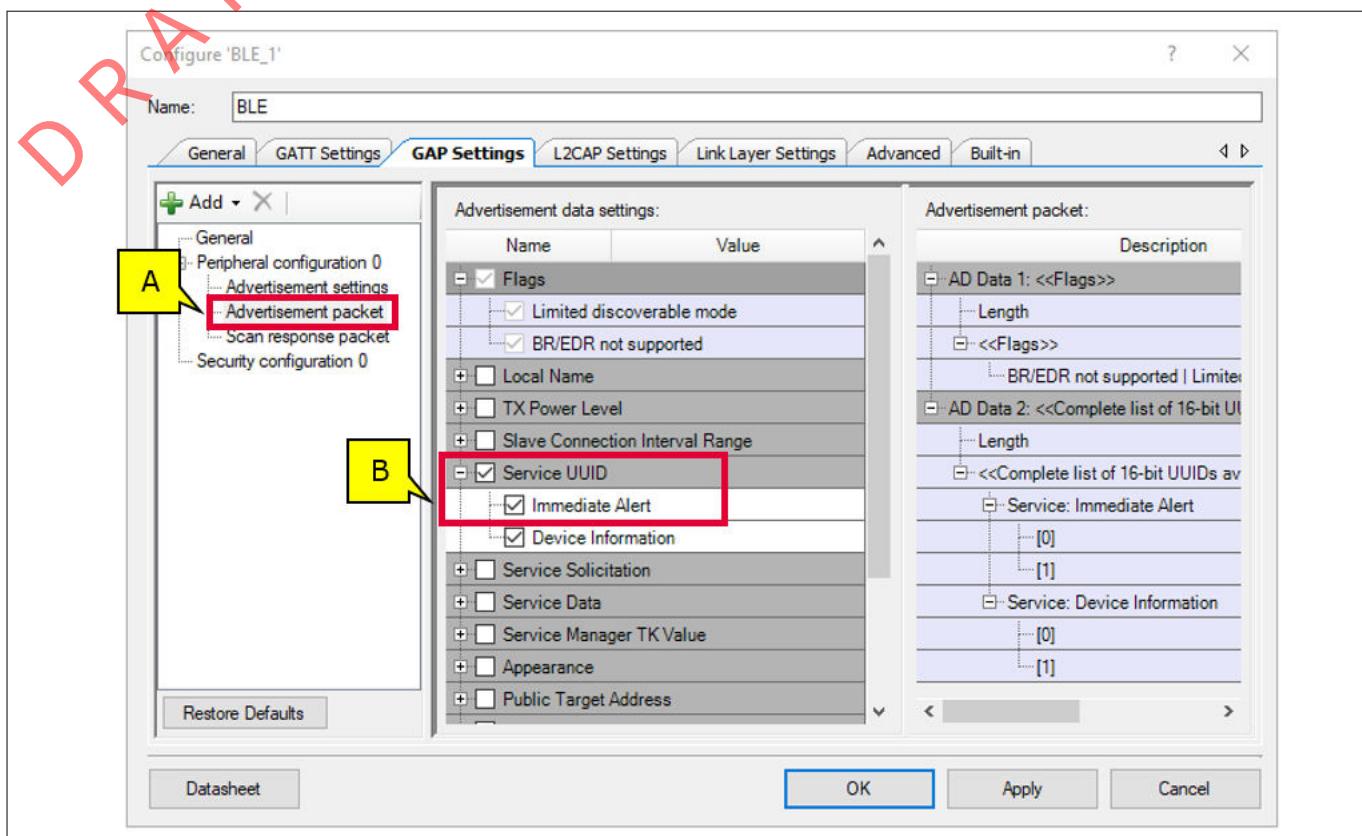


Figure 92 Bluetooth® Low Energy component GAP advertisement packet settings

- Specify scan response packet settings

In this step, you specify the configuration for the Scan response packet. [Figure 93](#) shows the result. Note that as you add values, the structure and content of the scan response packet appears to the right of the configuration panel.

- Click the **Scan response packet** item in the left menu. The panel appears
- Select **Local Name** to include that item in the response. The default setting of **Complete** is **OK**
- Select **TX Power Level** to include that item in the packet
- Select **Appearance** to include that item in the packet

5 PSoC™ 6 application notes

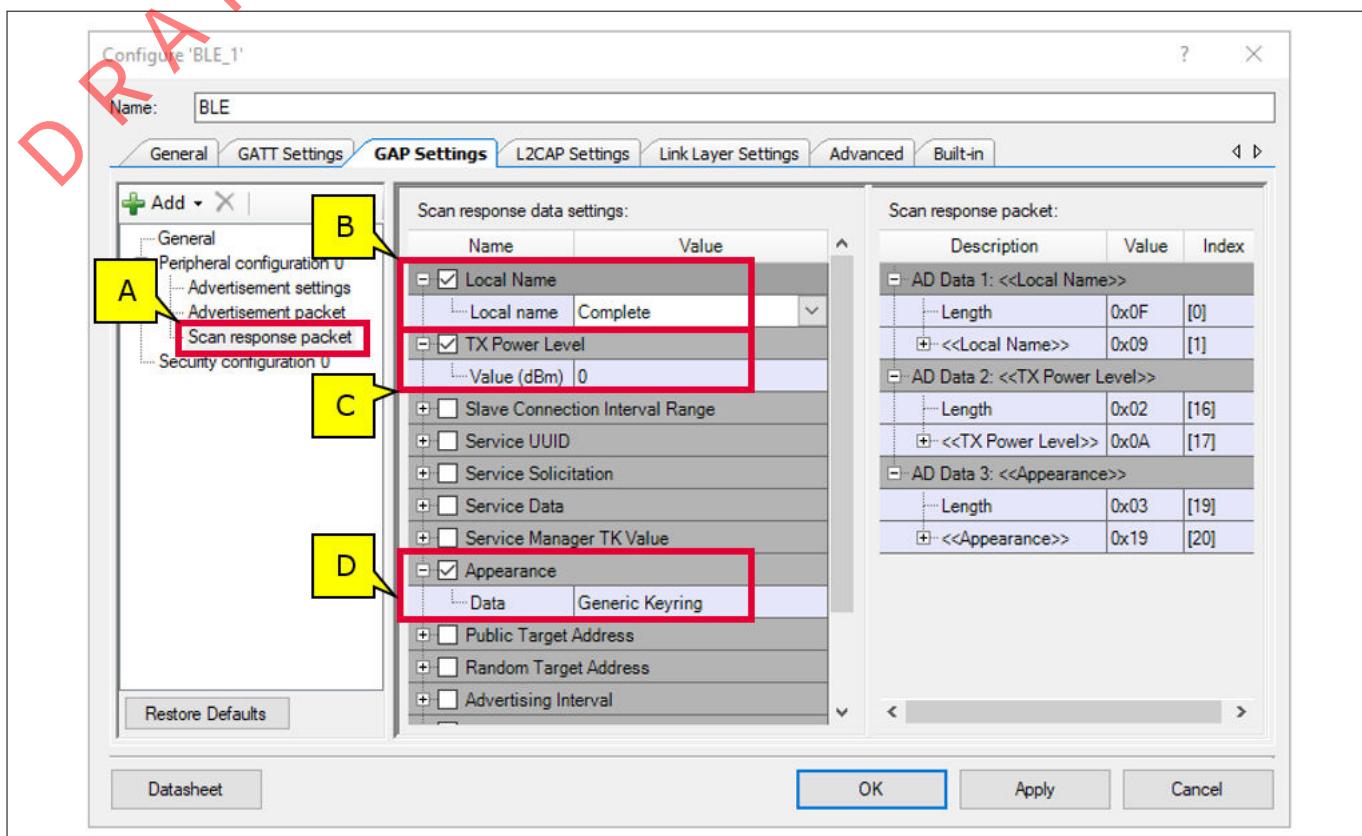


Figure 93 Bluetooth® Low Energy component GAP scan response packet

- Specify security configuration settings

In this step, you configure security settings for the device. It uses a configuration that does not require authentication, encryption, or authorization for data exchange. See [Figure 94](#).

- Click the **Security configuration 0** item in the left menu. The panel appears
- Confirm that **Security mode** is **Mode 1** and **Security level** is **No Security**. If not, modify the settings
- Set **IO capabilities** to **No Input No Output**
- Set **Bonding requirement** to **No Bonding**
- Click **OK** to complete the configuration of the Bluetooth® Low Energy Component Bluetooth® Low Energy Component

5 PSoC™ 6 application notes

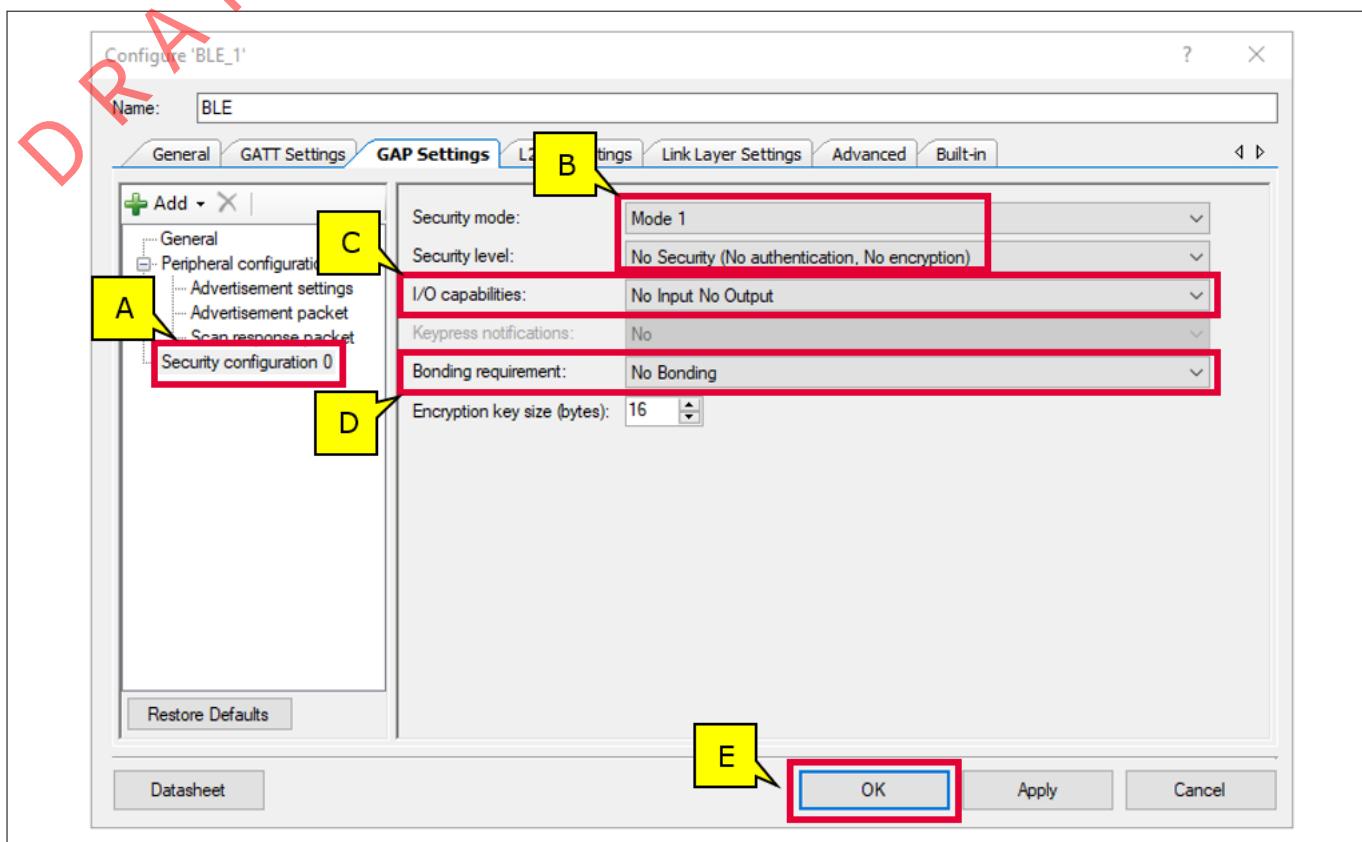


Figure 94 Bluetooth® Low Energy component GAP security settings

In this design, all other settings use default values, including all options in the **LDCAP Settings**, **Link Layer Settings**, and **Advanced** tabs.

See the Component datasheet to learn the significance of each setting.

- **Configure the MCWDT Component to trigger an interrupt**

In this step, you configure the Multi-Counter Watchdog (MCWDT) Component to trigger an interrupt every 250 ms (4 Hz). The clock source of the MCWDT is the low frequency clock (LFCLK). The LFCLK's source is the Internal Low Speed Oscillator (ILO) by default. The design will use this interrupt to blink the Alert LED at 2 Hz when a MILD alert is received. See [Figure 95](#). Double click the Component placed on the schematic and configure as below.

- Change the **Name** to **MCWDT**
- For Counter0, change the **Match** value to **7999**, **Mode** to **Interrupt**. Set the **Clear on Match** field to **Clear on match**
- Click **OK** to complete the configuration of the MCWDT Component

5 PSoC™ 6 application notes

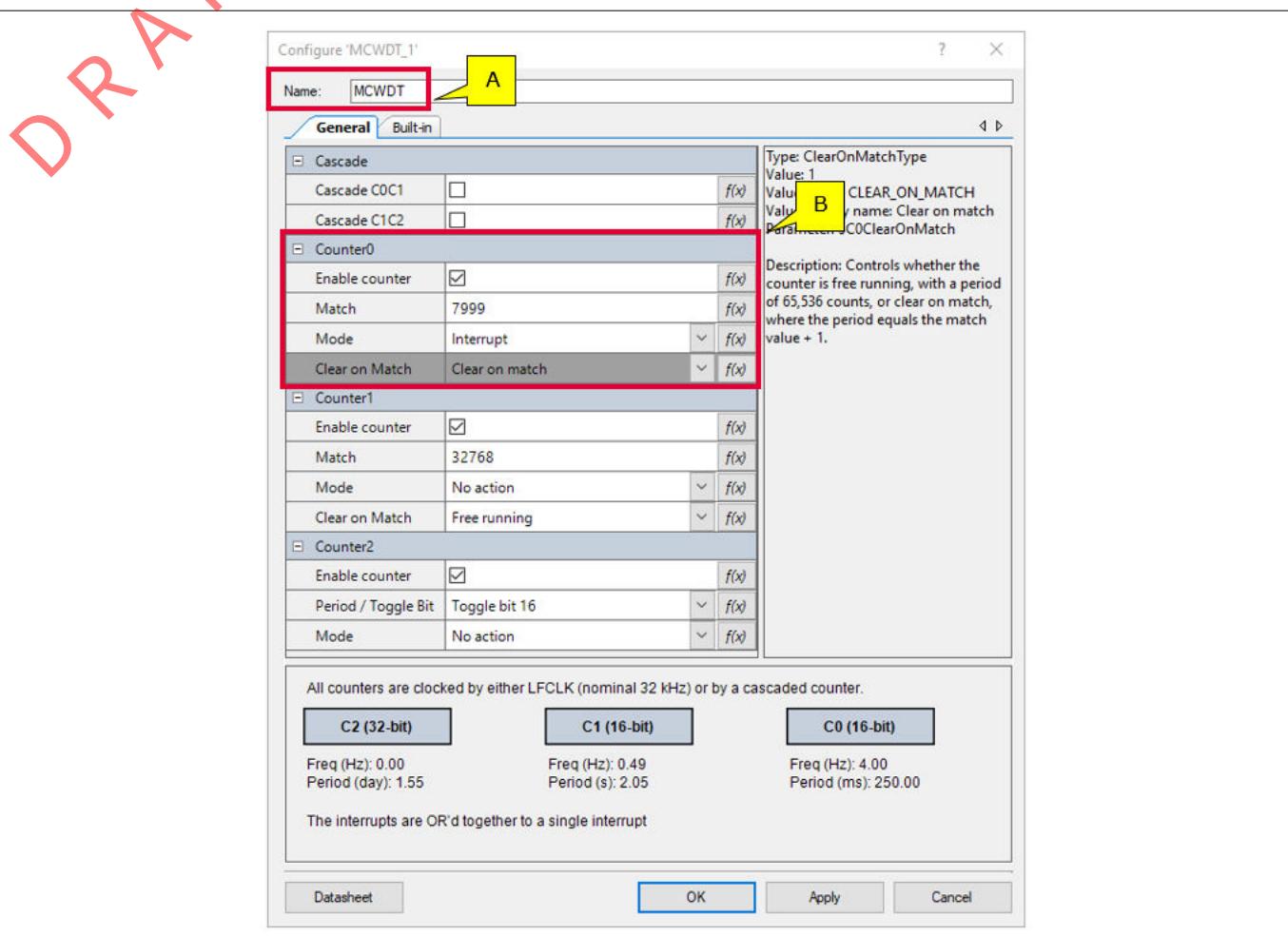


Figure 95 MCWDT_PDL settings

- Configure the interrupt Component to wake up the CM4 CPU in Deep Sleep mode

In this step, you configure the SysInt Component to wake up the CM4 CPU. See [Figure 96](#).

- Change the Name to MCWDT_isr
- Select the interrupt to be Deep Sleep Capable
- Click OK to complete the configuration of the SysInt Component

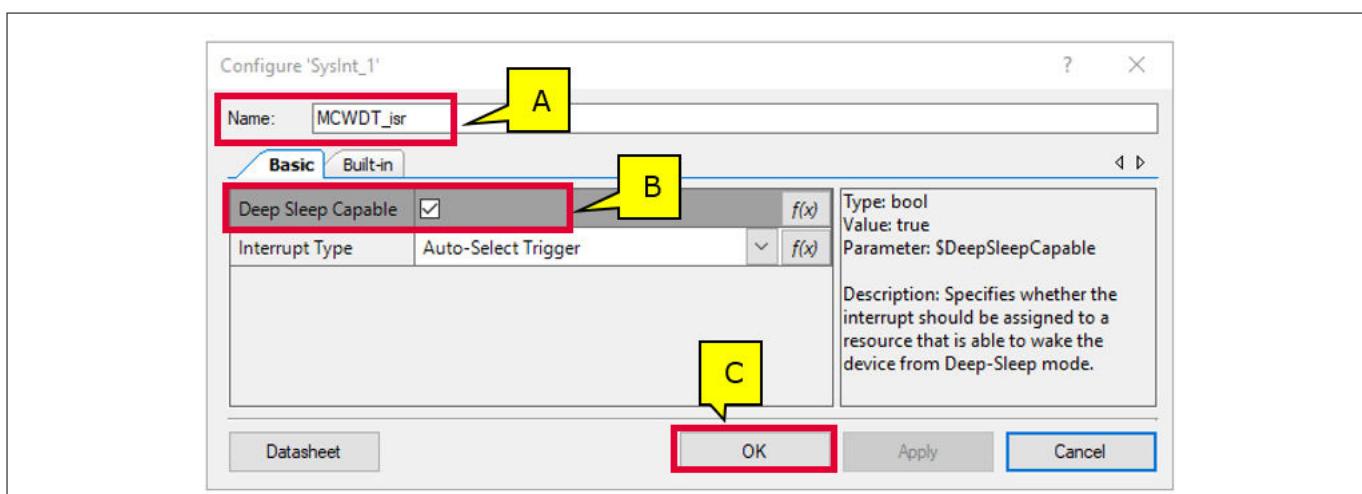


Figure 96 SysInt_PDL settings

~~5 PSoC™ 6 application notes~~

As a final step, connect the interrupt output of the MCWDT Component to MCWDT_isr Component input. This routes the watchdog interrupt to the CPU (the selection of the CM4 CPU for this interrupt will be set in the system interrupt configuration in a later step). In the schematic, use the wire tool button or press the **W** key to start wiring the components.

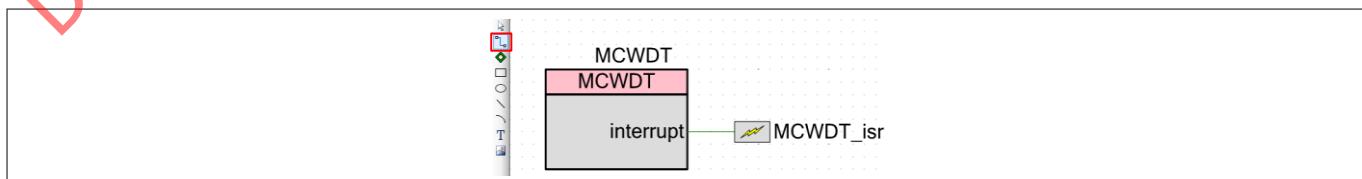


Figure 97 Connect MCWDT peripheral interrupt to CM4 CPU

- **Set the physical pins for each pin component**

One task remains to complete the design. You must associate each component with the required physical pins on the device. The choice of which pin to use is driven by the board design. You can find this information in the kit schematic. [Figure 98](#) shows the end result of this step.

To set a pin, type either the port number or pin number in the corresponding field, or use the drop-down menu to pick the port or pin. Typically, the port number is used instead of the pin number since these names are independent of the specific package being used.

- Open the pin selector

In the **Workspace Explorer** pane, double-click the **Pins** item under the Design wide resources. The pin selector for this device appears.

- Set each pin as shown in [Table 8](#)

Table 8 Physical pin assignments for CY8CKIT-062-BLE

Pin component name	Port name
UART_DEBUG:rx	P5[0]
UART_DEBUG:tx	P5[1]
Advertising_LED	P1[1]
Alert_LED	P11[1]
Disconnect_LED	P0[3]
Hibernate_Wakeup_SW	P0[4]

5 PSoC™ 6 application notes



Figure 98 Pin assignment

- **System clock configuration**

The design uses default values for the high frequency system clock settings. Although you do not modify high frequency clocks for this design, you should know how PSoC™ Creator manages them. If you are working with your own board, you may need to modify these clocks.

In this step, you set the low frequency clock source to be the accurate watch crystal oscillator (WCO) on the board. This clock is used by the Bluetooth® Low Energy Subsystem (BLESS) for timing purposes.

- In the **Workspace Explorer** pane, double-click the **Clocks** item under **Design Wide Resources (CE212736.cydwr)**. The list of clocks appears
- Click **Edit Clock...** and the **Configure System Clocks** dialog appears
- Here you can see the clock tree, and modify the clocks as required. Note that there are tabs for different types of clocks such as **Source Clocks**, **FLL/PLL**, **High Frequency Clocks**, and **Miscellaneous Clocks**
- Click the **Source Clocks** tab
- Enable the BLE ECO by selecting the checkbox in the **AltHF BLE ECO** block. The parameters should match the crystal used on the board. For the CY8CKIT-062-BLE kit, the **ECO Frequency** is **32 MHz**, and **Accuracy** is **±50 ppm**. There are no load caps on the board and hence use the minimum specified **Load cap (pF)** of **9.900**. Ensure that the **Startup Time (μs)** is **785 μs** for a quick startup of the ECO crystal
- Enable the WCO clock by selecting the checkbox in the **WCO (32.768 kHz)** block
- Click the **Miscellaneous Clocks** tab
- Select **WCO** to be the source for **LFClk**
- Select **WCO** to be the source for **BakClk**

5 PSoC™ 6 application notes

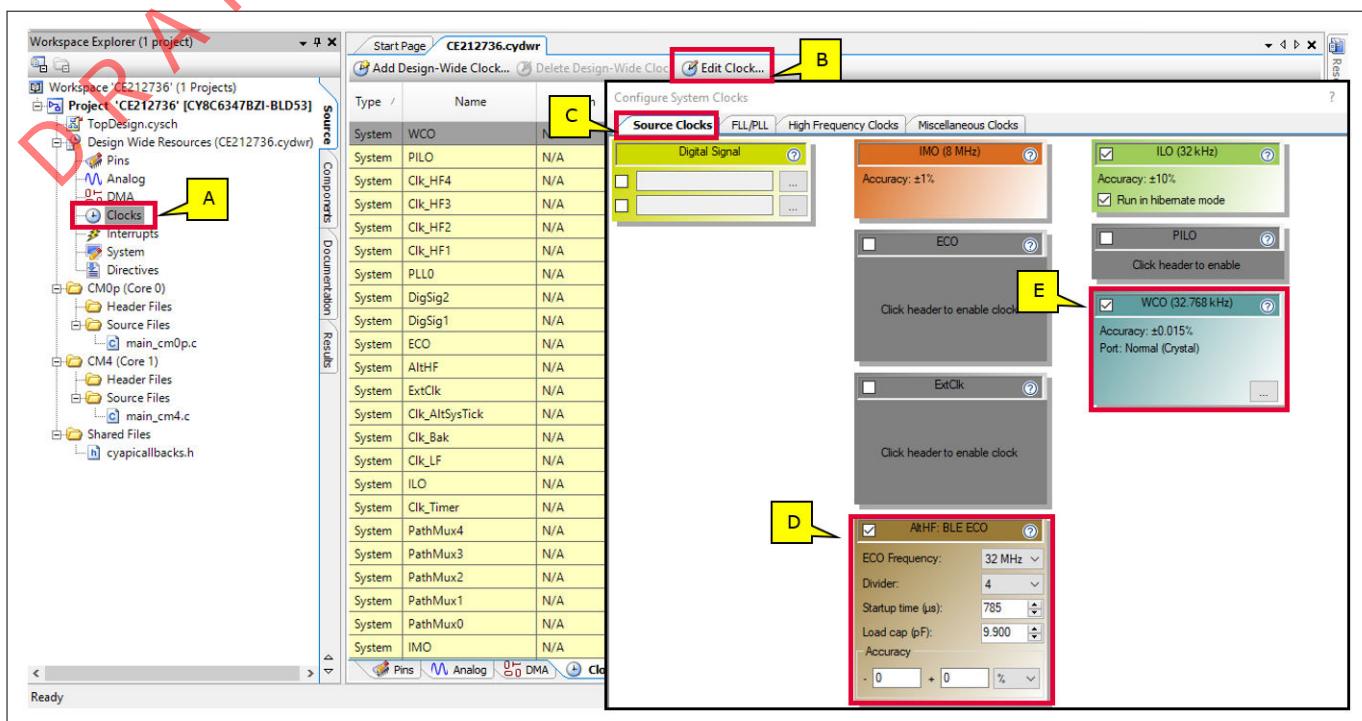


Figure 99 Clock configuration - source clocks

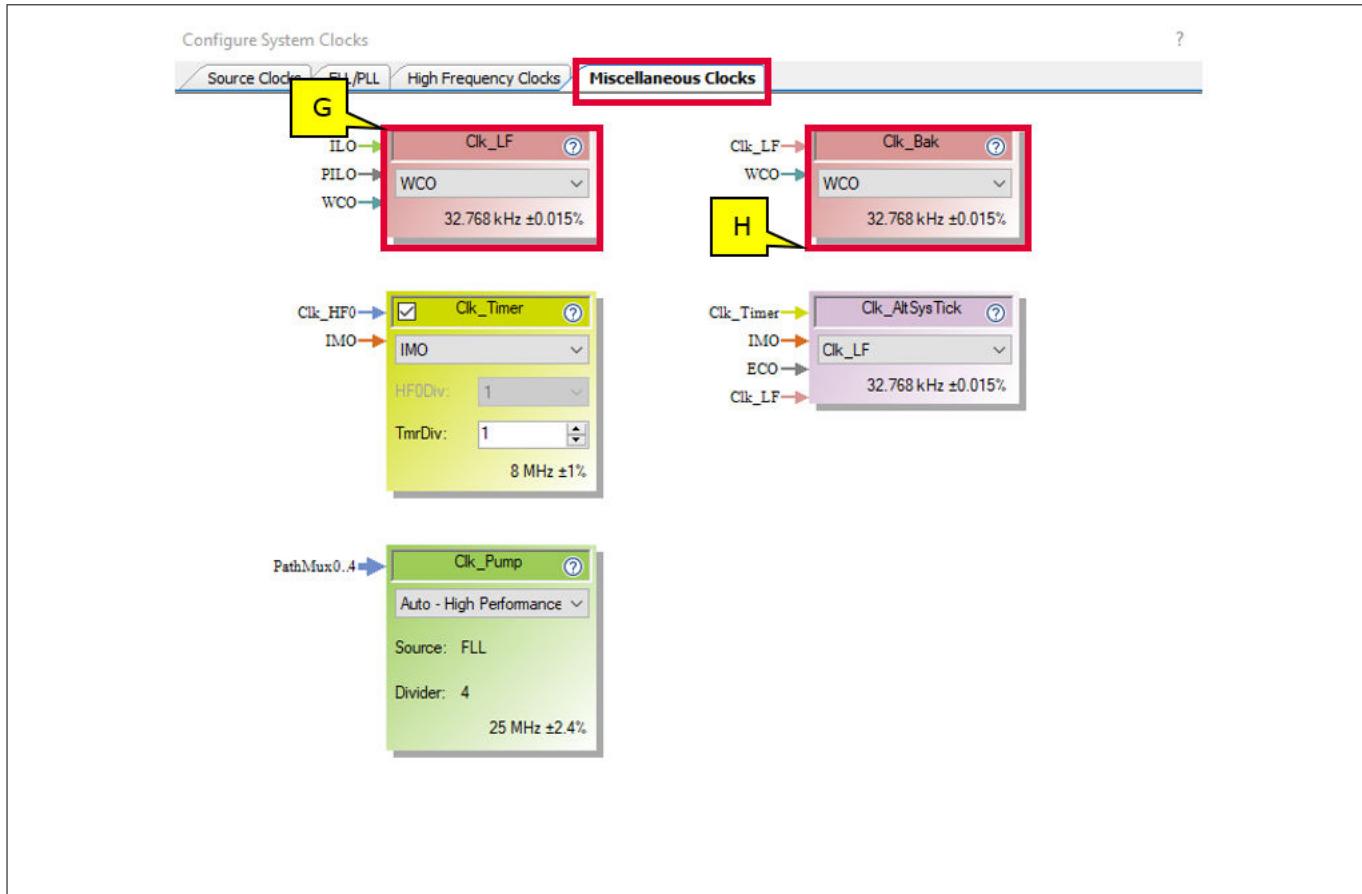


Figure 100 Clock configuration - miscellaneous clocks

- **System interrupt configuration**

In this step, you configure the system interrupts. See [Figure 101](#).

5 PSoC™ 6 application notes

- In the **Workspace Explorer** pane, double-click the **Interrupts** item under **Design Wide Resources**. The list of interrupts appears
- Confirm that the **BLE_bless_isr** is enabled for the CM0+ CPU, the priority is set to **1**, and vector is set to **3**. This ensures that the Bluetooth® LE controller interrupts are handled by CM0+ at the highest priority, and in Deep Sleep mode as well
- Enable **MCWDT_isr** for CM4
- Disable the **UART_DEBUG_SCB_IRQ** interrupt for both cores
- Deselect the corresponding checkboxes. You can ignore the interrupt related to UART_DEBUG Component because the design does not use it

The interrupt numbers are generated automatically by PSoC™ Creator when you generate the code in [Part 3. Generate source code](#).

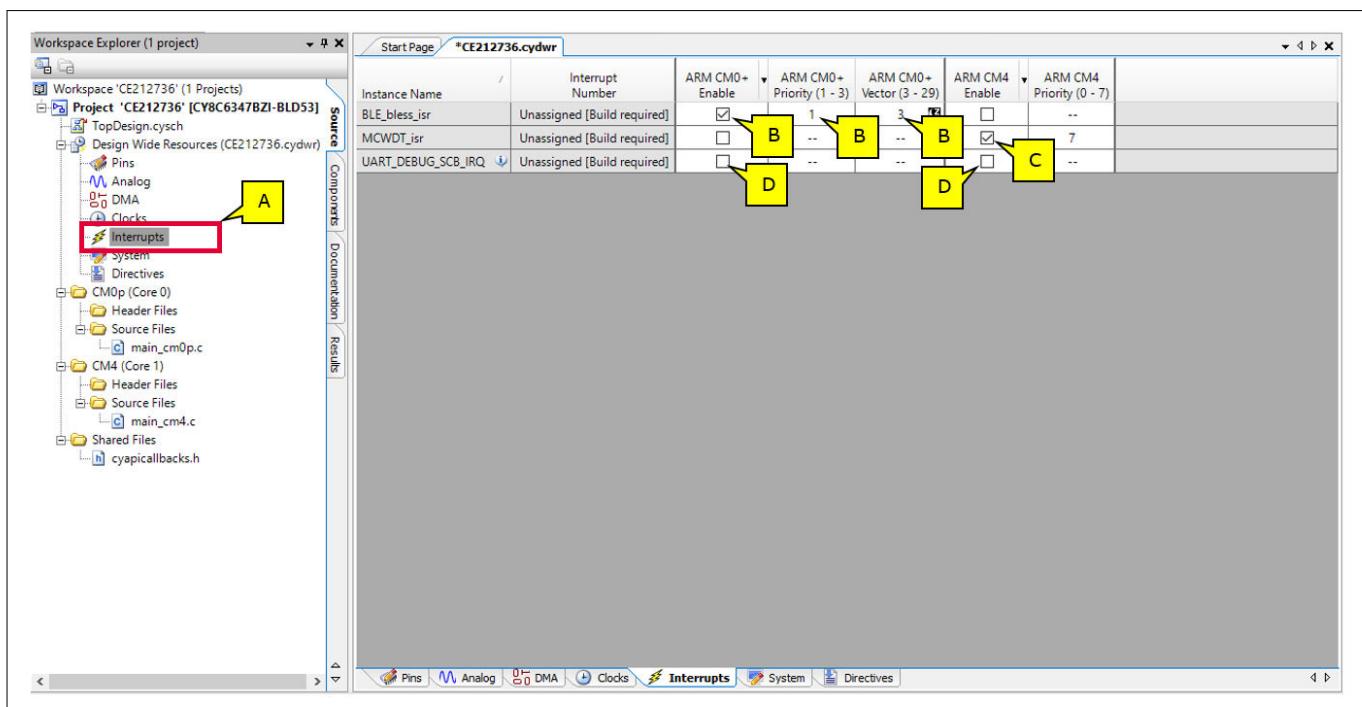


Figure 101 **Interrupt configuration**

The next part in the development process is to generate code.

Note: *This exercise does not detail how to export your work to a target IDE. However, if you wish to use a target IDE this is the point in the workflow where you would ensure that the correct target IDE is selected, before you generate code. See [Support for other IDEs](#).*

5.3.5.6 Part 3. Generate source code

PSoC™ Creator generates source code based upon the design. The recommended workflow is to generate code before writing firmware. PSoC™ Creator will automatically create macros, constants, and API calls that you may then use in your firmware.

This part of the exercise is very simple, and the path is the same for everyone.

5 PSoC™ 6 application notes

Path	Working from scratch code example as reference only	Using code example new to PSoC™ Creator or Bluetooth® LE	Using code example familiar with PSoC™ Creator and Bluetooth® LE
Actions	Perform the one step	Perform the one step	Perform the one step

• Generate the application

Choose **Build > Generate application**. PSoC™ Creator generates source code based on the design and puts the files in the *Generated_Source* folder. See [Figure 102](#). PSoC™ Creator will alert you to errors or problems that may occur. If you are working from scratch and encounter errors, revisit the configuration steps in [Part 2. Implement the design](#) to ensure you have performed them correctly.

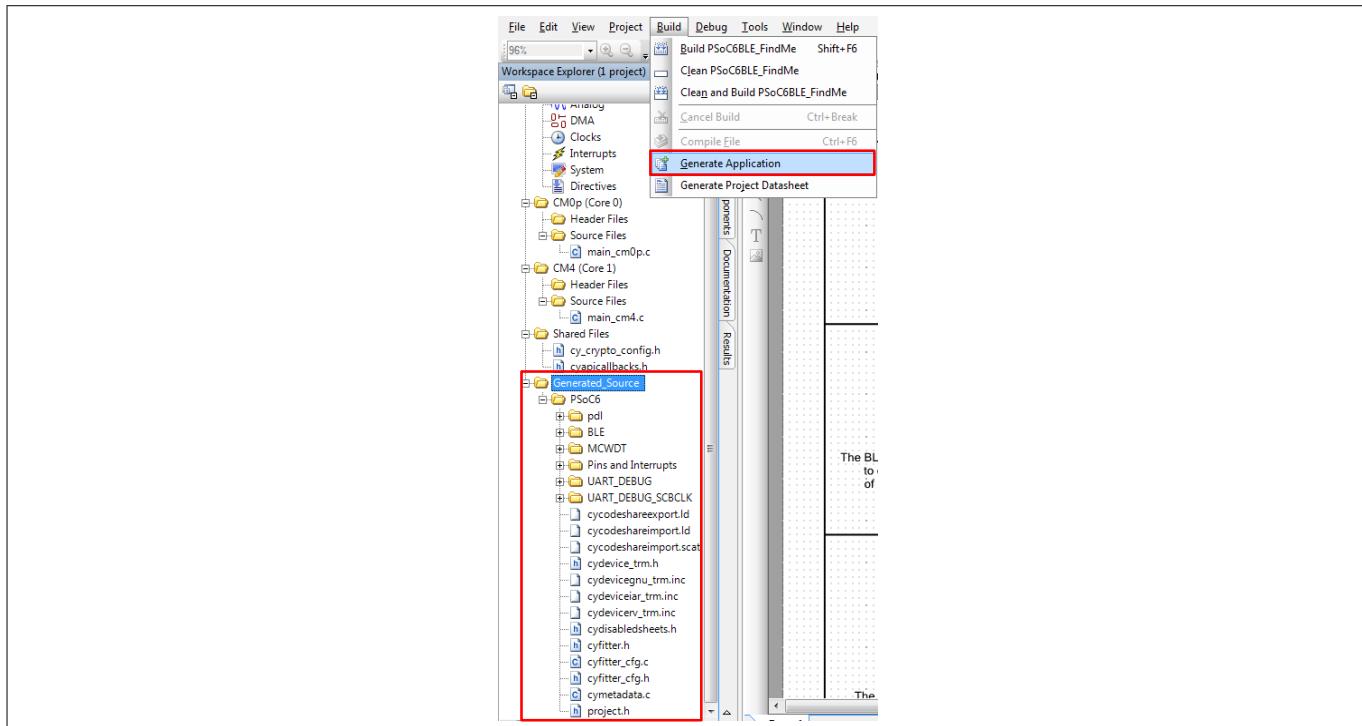


Figure 102 Generate application

Background: PSoC™ 6-BLE is a dual-CPU platform. You can target firmware to run either on CM4 or CM0+. You set this at the source file level by accessing the file properties. Right-click a source file and select **Properties**.

[Figure 102](#) shows the **Properties** dialog window. This code example targets the CM4 core.

By default, the *main_cm0p.c* file is targeted to CM0+ and the *main_cm4.c* file is targeted to CM4. You do not need to modify the properties for any other file. They are already set in the code example.

By convention, files targeted to run on the CM0+ are in the *CM0p* folder and files targeted to run on CM4 are in the *CM4* folder, but the properties must also be set appropriately – just putting a file in the correct folder does not cause it to run on a specific core.

5 PSoC™ 6 application notes

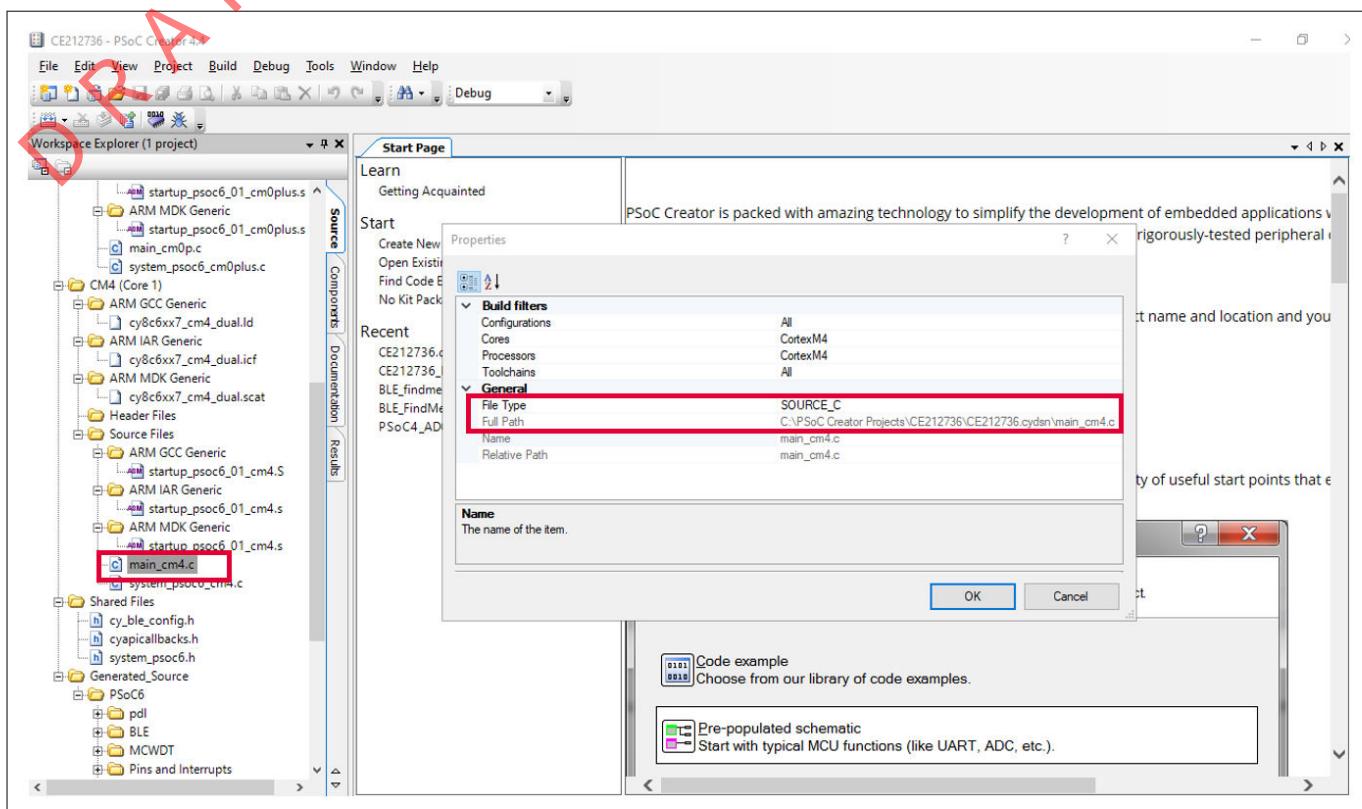


Figure 103 Setting target processor for a source C file

5.3.5.7 Part 4. Write the firmware

At this point in the development process you have created a project, implemented a hardware design, and generated code. In this part, you examine the firmware that implements Bluetooth® LE functionality in the application.

The firmware must accomplish several tasks to implement a Bluetooth® LE standard profile application, including:

- Perform system initialization
- Implement a BLE stack event handler
- Implement a BLE service-specific event handler
- Provide the main loop
- Implement low-power performance (optional)

The steps in this part discuss the firmware for the design that you configured in [Part 2. Implement the design](#). The steps do not examine every single line of code but point out important elements in the code that implement significant functionality.

Path	Working from scratch code example as reference only	Using code example new to PSoC™ Creator or Bluetooth® LE	Using code example familiar with PSoC™ Creator and Bluetooth® LE
Actions	Perform all steps	Skip step one Perform all other steps	Skip this part if you wish. Jump to Part 5. Build the project, program the Device

~~5 PSoC™ 6 application notes~~

The code example has all the required code. If you are working from scratch, in step one you copy the source files from the code example project. If you are using the code example, those files are already in your project, so you can skip step one.

~~• Add files to your project~~

If you are using the code example, you can skip this step. The code example already has the required source files.

If you are working from scratch, the required source code files are not in your project.

- Locate the *CE212736.cydsn* folder, which contains the source files for the code example. The folder is in the Code Example workspace archive that you downloaded earlier
- Copy these files from the *CE212736.cydsn* folder to your project's *.cydsn* folder. Replace any existing files
 - *BLEFindMe.h*
 - *debug.h*
 - *LED.h*
 - *BLEFindMe.c*
 - *debug.c*
 - *main_cm0p.c*
 - *main_cm4.c*
- Add these files to your project. You can do this by dragging them from the *Windows* folder onto the Workspace explorer, and dropping them in the *CM4* folder location in the workspace. See [Figure 104](#)
 - *BLEFindMe.h*
 - *debug.h*
 - *LED.h*
 - *BLEFindMe.c*
 - *debug.c*

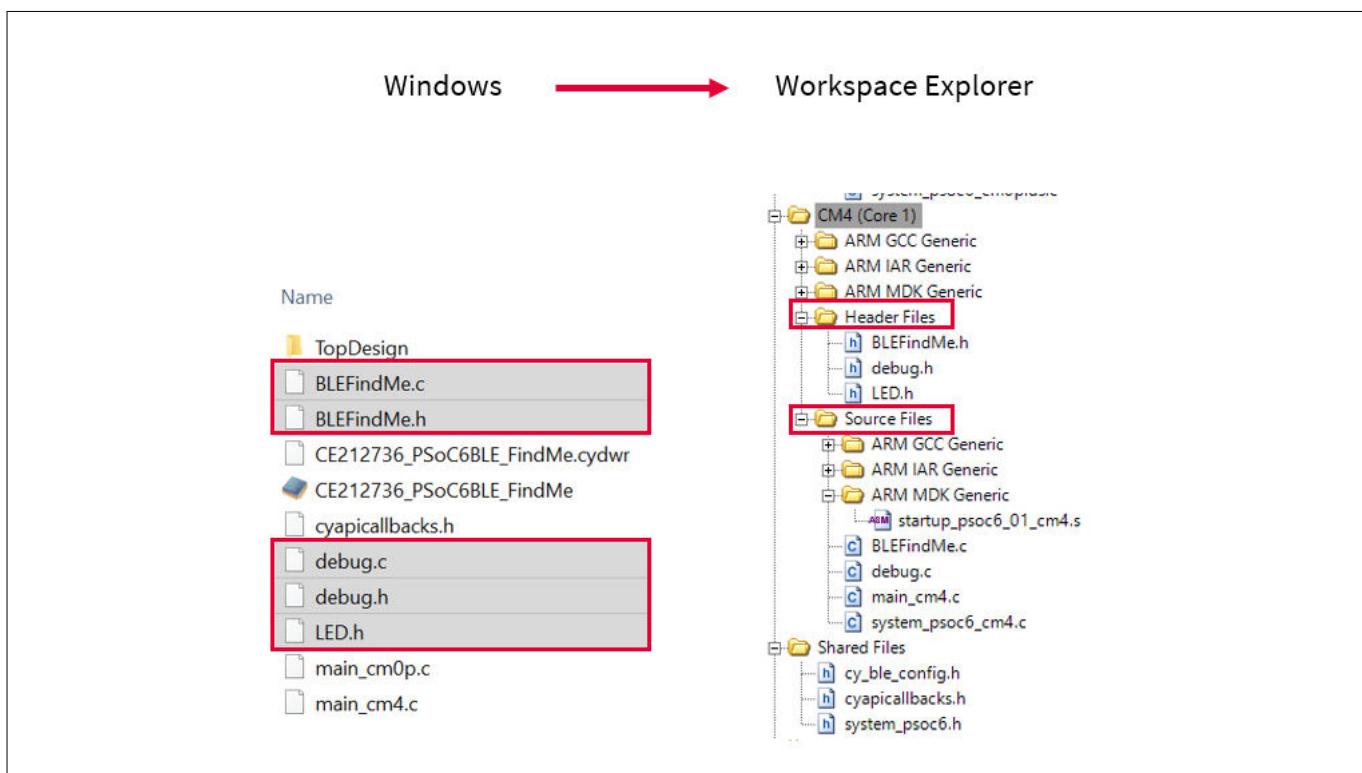


Figure 104 **Add files to your project**

5 PSoC™ 6 application notes

~~DRAFT~~

You do not have to add *main_cm0p.c* or *main_cm4.c*. The project has these files by default. You just replaced them in the *project* folder, so the project will use the newer version you just copied.

- **Initialize the system**

In the remaining steps, we examine code in the *main_cm0p.c* and *main_cm4.c* file. The code snippets frequently have the debugging print statements removed for clarity. See the actual source file for a complete understanding of the code.

Figure 105 shows the steps in the process of initializing the system. When the PSoC™ 6-BLE device is reset, the firmware first performs system initialization, which includes setting up the CPU cores for execution, enabling global interrupts, and enabling other Components used in the design. The initialization is split across the CPU cores. The CM0p CPU comes out of reset and attempts to start the Bluetooth® LE controller part. If it is successful, the CM0p CPU then enables CM4 CPU. The CM4 CPU will start the Bluetooth® LE host part and register the necessary application side handler functions.

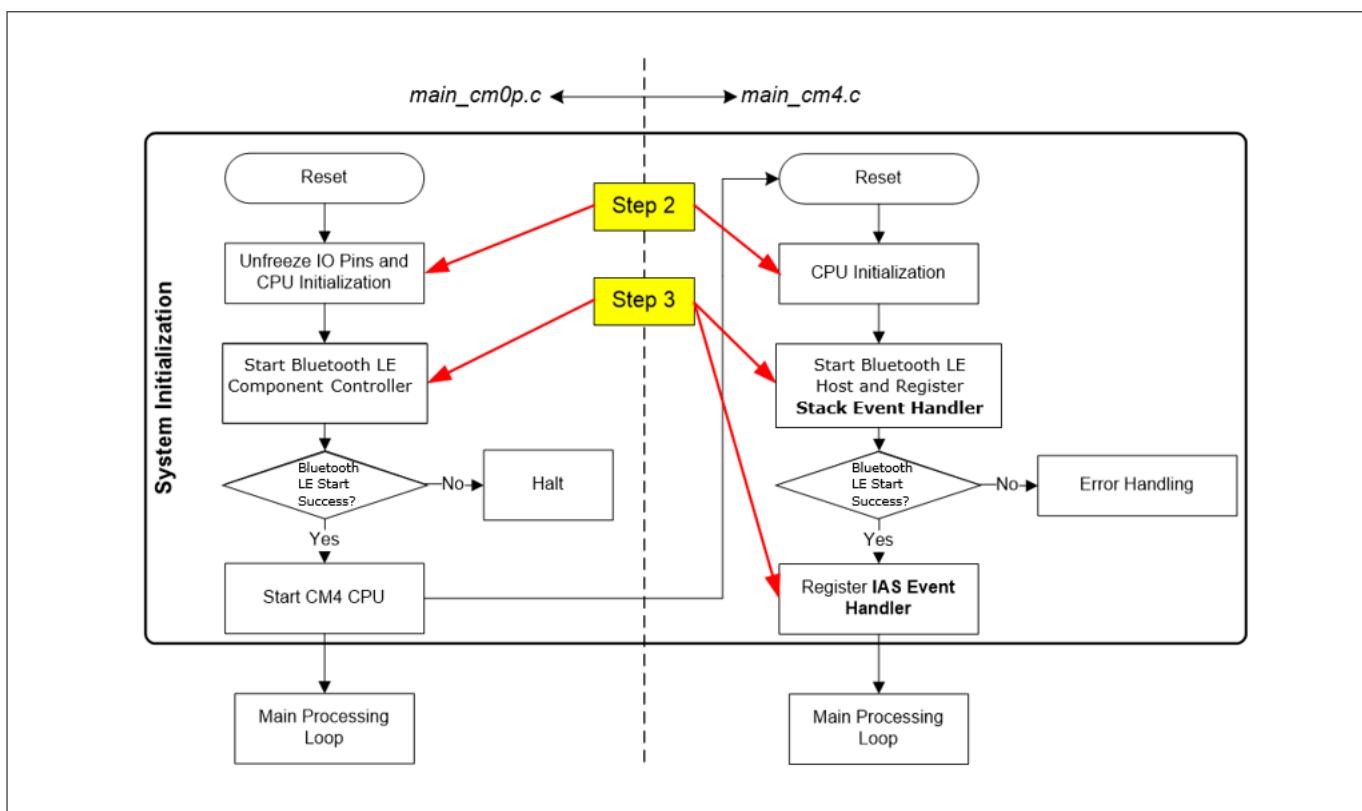


Figure 105 System initialization flowchart – CM4

The code in *main_cm0p.c* declares a local variable to hold the return value from Bluetooth® LE API calls. The key task is to enable the Bluetooth® LE controller, and set up the CM4 for the application code to run. In the main

5 PSoC™ 6 application notes

loop, the CM0p CPU processes the Bluetooth® LE events pending on the controller. In case of no pending events, the CM0p enters Deep Sleep mode.

~~DRAFT~~

```
int main(void)
{
    cy_en_ble_api_result_t      apiResult;
    __enable_irq(); /* Enable global interrupts. */
    /* Unfreeze IO if device is waking up from hibernate */
    if(Cy_SysPm_GetIoFreezeStatus())
    {
        Cy_SysPm_IoUnfreeze();
    }
    /* Start the Controller portion of BLE. Host runs on the CM4 */
    apiResult = Cy_BLE_Start(NULL);
    if(apiResult == CY_BLE_SUCCESS)
    {
        /* Enable CM4 only if BLE Controller started successfully.
         * CY_CORTEX_M4_APPL_ADDR must be updated if CM4 memory layout
         * is changed. */
        Cy_SysEnableCM4(CY_CORTEX_M4_APPL_ADDR);
    }
    else
    {
        /* Halt CPU */
        CY_ASSERT(0u != 0u);
    }
    for(;;)
    {
        /* Place your application code here. */
        /* Put CM0p to deep sleep. */
        Cy_SysPm_DeepSleep(CY_SYSPM_WAIT_FOR_INTERRUPT);

        /* Cy_Ble_ProcessEvents() allows BLE stack to process pending events */
        /* The BLE Controller automatically wakes up host if required */
        Cy_BLE_ProcessEvents();
    }
}
```

The code in `main_cm4.c` initializes the key Components to be used by the CM4 and continuously runs a Bluetooth® LE application process. The `BleFindMe_Init()` routine initializes and starts all the Components, including setting up the CM4 interrupt. It performs the key task of enabling the Bluetooth® LE host.

5 PSoC™ 6 application notes

In the application process, the CM4 CPU processes the Bluetooth® LE events pending on the host. If there are no events pending, the CM4 enters Deep Sleep low-power mode.

~~DRAFT~~

```
int main(void)
{
    __enable_irq(); /* Enable global interrupts. */

    /* Initialize BLE */
    BleFindMe_Init();

    for(;;)
    {
        BleFindMe_Process();
    }
}
```

- **Start the Bluetooth® Low Energy component and register the event handlers**

After the CPUs are initialized, the firmware initializes the Bluetooth® Low Energy component, which sets up the complete Bluetooth® LE subsystem. The BleFindMe_Init() subroutine handles the work. To focus on the key tasks, debug print statements have been removed. Examine the source file to see the full code.

As a part of the Bluetooth® Low Energy Bluetooth® Low Energy component initialization, you must pass the event handler function, which will be called by the Bluetooth® LE stack to notify of pending events. If the Bluetooth® Low Energy component initializes successfully, the firmware registers a second event handler for events specific to the IAS.

The code uses PDL API function calls to configure the application. First it starts the Bluetooth® Low Energy Component. The parameter is the address of the stack event handler function.

The code also gets the stack version. In case the debug port is enabled, the version is printed in the serial communication window.

5 PSoC™ 6 application notes

~~DRAFT~~

It then registers the IAS event handler to handle Immediate Alert Service related events. Finally, it configures and enables the MCWDT to trigger interrupts once every 250 ms.

```

void BleFindMe_Init(void)
{
    cy_en_ble_api_result_t apiResult;
    cy_stc_ble_stack_lib_version_t stackVersion;

    /* Configure switch SW2 as hibernate wake up source */
    Cy_SysPm_SetHibWakeupSource(CY_SYSPM_HIBPIN1_LOW);

    /* Start Bluetooth® Low Energy Component and register generic event handler */
    apiResult = Cy_BLE_Start(StackEventHandler);
    apiResult = Cy_BLE_GetStackLibraryVersion(&stackVersion);

    /* Register IAS event handler */
    Cy_BLE_IAS_RegisterAttrCallback(IasEventHandler);

    /* Enable 4 Hz free-running MCWDT counter 0*/
    /* MCWDT_config structure is defined by the MCWDT_PDL component based
       on parameters entered in the customizer. */
    Cy_MCWDT_Init(MCWDT_HW, &MCWDT_config);
    Cy_MCWDT_Enable(MCWDT_HW, CY_MCWDT_CTR0, 93 /* 2 LFCLK cycles */);

    /* Unmask the MCWDT counter 0 peripheral interrupt */
    Cy_MCWDT_SetInterruptMask(MCWDT_HW, CY_MCWDT_CTR0);

    /* Configure ISR connected to MCWDT interrupt signal*/
    /* MCWDT_isr_cfg structure is defined by the SYSINT_PDL component based
       on parameters entered in the customizer. */
    Cy_SysInt_Init(&MCWDT_isr_cfg, &MCWDT_Interrupt_Handler);

    /* Clear CM4 NVIC pending interrupt for MCWDT */
    NVIC_ClearPendingIRQ(MCWDT_isr_cfg.intrSrc);

    /* Enable CM4 NVIC MCWDT interrupt */
    NVIC_EnableIRQ(MCWDT_isr_cfg.intrSrc);
}

```

- **Implement the stack event handler**

The Bluetooth® LE stack within the Bluetooth® Low component generates events. These events provide status and data to the application firmware through the Bluetooth® LE stack event handler. [Figure 106](#) shows a simplified flowchart representing certain events.

5 PSoC™ 6 application notes

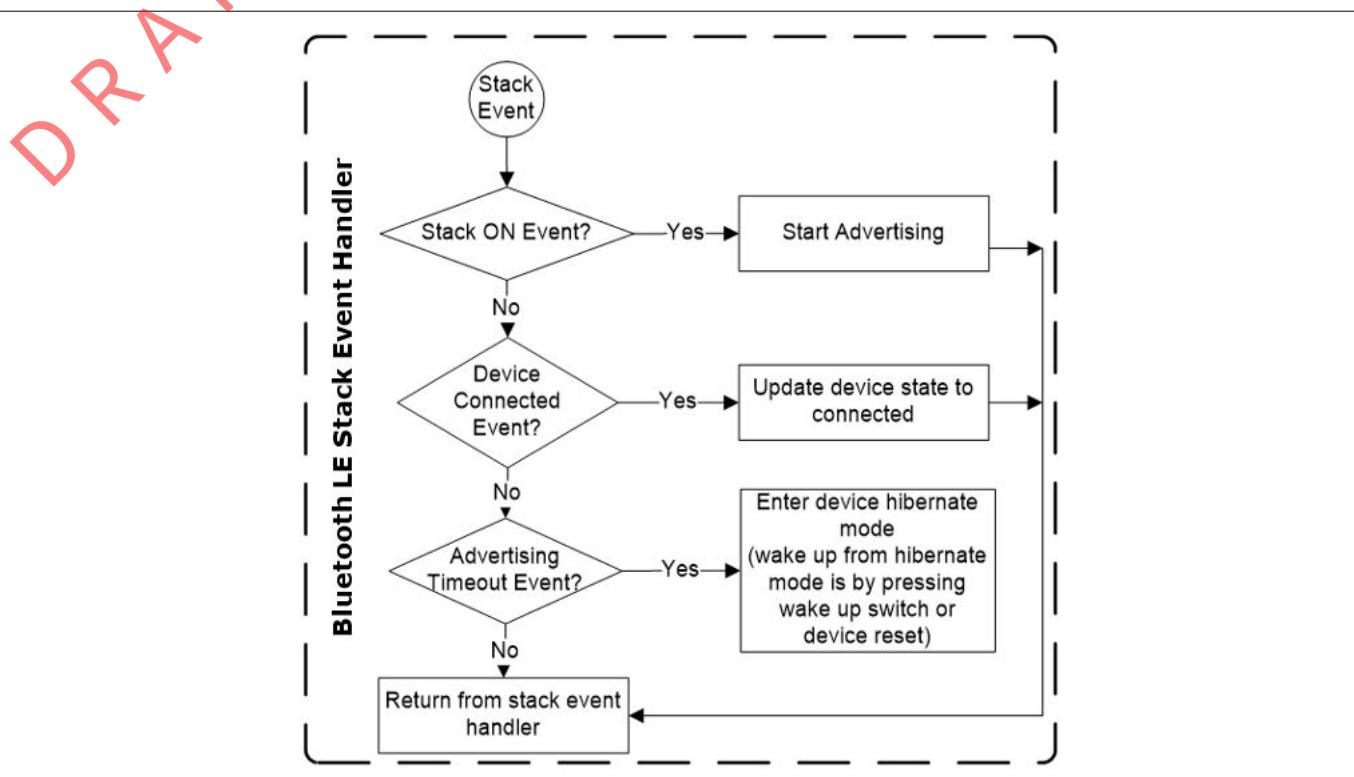


Figure 106 BLE stack event handler flowchart

The event handler must handle a few basic events from the stack. For the Find Me Target application in this code example, the BLE stack event handler must process the events described in [Table 9](#). The actual code recognizes and responds to additional events, but they are not mandatory for this application.

Table 9 Bluetooth® LE stack events

Bluetooth® LE stack event name	Event description	Event handler action
CY_BLE_EVT_STACK_ON	Bluetooth® LE stack initialization is completed successfully	Start advertisement and reflect the advertisement state on the LED
CY_BLE_EVT_GAP_DEVICE_DISCONNECTED	Bluetooth® LE link with the peer device is disconnected	Restart advertisement and reflect the advertisement state on the LED
CY_BLE_EVT_GAP_DEVICE_CONNECTED	Bluetooth® LE link with the peer device is established	Update the BLE link state on the LED
CY_BLE_EVT_GAPP_ADVERTISEMENT_START_STOP	Bluetooth® LE stack advertisement start/stop event	Shutdown the Bluetooth® LE stack
CY_BLE_EVT_HARDWARE_ERROR	Bluetooth® LE hardware error	Update the LED status to reflect a hardware error and halt the CPU
CY_BLE_EVT_STACK_SHUTDOWN_COMPLETE	BLE stack has been shut down	Configure the device in Hibernate mode and wait for event on wakeup pin

The code snippets show two examples of how the event handler responds to an identified event. See the actual source code for a complete understanding.

In this snippet, the handler responds to the “advertisement start/stop” event. The code toggles the LEDs appropriately. If advertisement has started, the advertisement LED turns on. The disconnect LED turns off,

5 PSoC™ 6 application notes

~~DRAFT~~
because the device started advertisement and is ready for a connection. If advertising is stopped, the code sets the LEDs appropriately, and sets a flag to enter Hibernate mode.

```
/* This event indicates peripheral device has started/stopped advertising */
case CY_BLE_EVT_GAPP_ADVERTISEMENT_START_STOP:
    DEBUG_PRINTF("CY_BLE_EVT_GAPP_ADVERTISEMENT_START_STOP: ");
    if(Cy_BLE_GetAdvertisementState() == CY_BLE_ADV_STATE_ADVERTISING)
    {
        DEBUG_PRINTF("Advertisement started \r\n");
        Cy_GPIO_Write(Advertising_LED_0_PORT, Advertising_LED_0_NUM, LED_ON);
        Cy_GPIO_Write(Disconnect_LED_0_PORT, Disconnect_LED_0_NUM, LED_OFF);
    }
    else if(Cy_BLE_GetAdvertisementState() == CY_BLE_ADV_STATE_STOPPED)
    {
        DEBUG_PRINTF("Advertisement stopped \r\n");
        Cy_GPIO_Write(Advertising_LED_0_PORT, Advertising_LED_0_NUM, LED_OFF);
        Cy_GPIO_Write(Disconnect_LED_0_PORT, Disconnect_LED_0_NUM, LED_ON);

        /* Advertisement event timed out before connection, shutdown BLE
         * stack to enter hibernate mode and wait for device reset event
         * or SW2 press to wake up the device */
        Cy_BLE_Stop();
    }
    break;
```

In this snippet, the handler responds to the “disconnected” event. It sets the LEDs correctly, and sets the Hibernate flag.

~~5 PSoC™ 6 application notes~~

~~DRAFT~~
These snippets give you a sense for how the event handler responds to events. Examine the actual function to see how each event is handled.

```

/* This event is generated when disconnected from remote device or
failed to establish connection. */

case CY_BLE_EVT_GAP_DEVICE_DISCONNECTED:
    if(Cy_BLE_GetConnectionState(appConnHandle) == CY_BLE_CONN_STATE_DISCONNECTED)
    {
        DEBUG_PRINTF("CY_BLE_EVT_GAP_DEVICE_DISCONNECTED %d\r\n", CY_BLE_CONN_STATE_DISCONNECTED);
        alertLevel = CY_BLE_NO_ALERT;
        Cy_GPIO_Write(Advertising_LED_0_PORT, Advertising_LED_0_NUM, LED_OFF);
        Cy_GPIO_Write(Disconnect_LED_0_PORT, Disconnect_LED_0_NUM, LED_ON);

        /* Enter into discoverable mode so that remote device can search it */
        apiResult = Cy_BLE_GAPP_StartAdvertisement(CY_BLE_ADVERTISING_FAST,
CY_BLE_PERIPHERAL_CONFIGURATION_0_INDEX);
        if(apiResult != CY_BLE_SUCCESS)
        {
            DEBUG_PRINTF("Start Advertisement API Error: %d \r\n", apiResult);
            ShowError();

            /* Execution does not continue beyond this point */
        }
        else
        {
            DEBUG_PRINTF("Start Advertisement API Success: %d \r\n", apiResult);
            Cy_GPIO_Write(Advertising_LED_0_PORT, Advertising_LED_0_NUM, LED_ON);
            Cy_GPIO_Write(Disconnect_LED_0_PORT, Disconnect_LED_0_NUM, LED_OFF);
        }
    }
    break;
}

```

- **Implement the service-specific event handler**

The Bluetooth® Low Energy Component also generates events corresponding to each of the services supported by the design. For the Find Me Target application, the Bluetooth® Low Energy Component generates IAS events that let the application know that the Alert Level characteristic has been updated with a new value. The event handler gets the new value and stores it in the variable alertLevel. The main loop toggles the alert LED based on the current alert level.

Figure 107 shows the IAS event handler flowchart.

5 PSoC™ 6 application notes

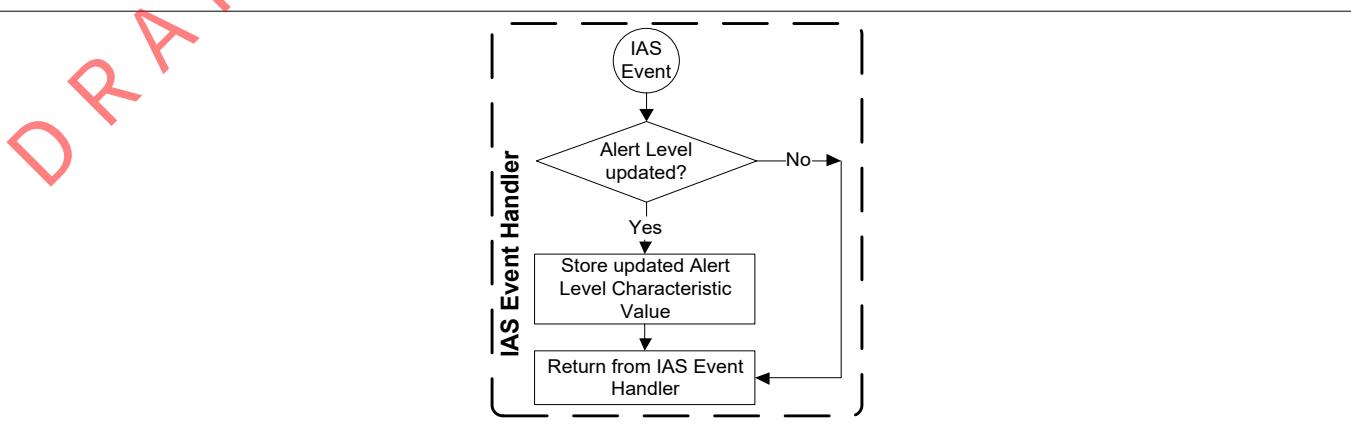


Figure 107 **Bluetooth® LE IAS event handler flowchart**

The code snippet shows how the firmware accomplishes this task.

```

void IasEventHandler(uint32 event, void *eventParam)
{
    /* Alert Level Characteristic write event */
    if(event == CY_BLE_EVT_IASS_WRITE_CHAR_CMD)
    {
        /* Read the updated Alert Level value from the GATT database */
        Cy_BLE_IASS_GetCharacteristicValue(CY_BLE_IAS_ALERT_LEVEL,
            sizeof(alertLevel), &alertLevel);
    }

    /* To remove unused parameter warning */
    eventParam = eventParam;
}
    
```

- **Process events as they occur (main loop)**

The main loop simply calls BleFindMe_Process(). [Figure 108](#) shows the BleFindMe_Process() flowchart.

5 PSoC™ 6 application notes

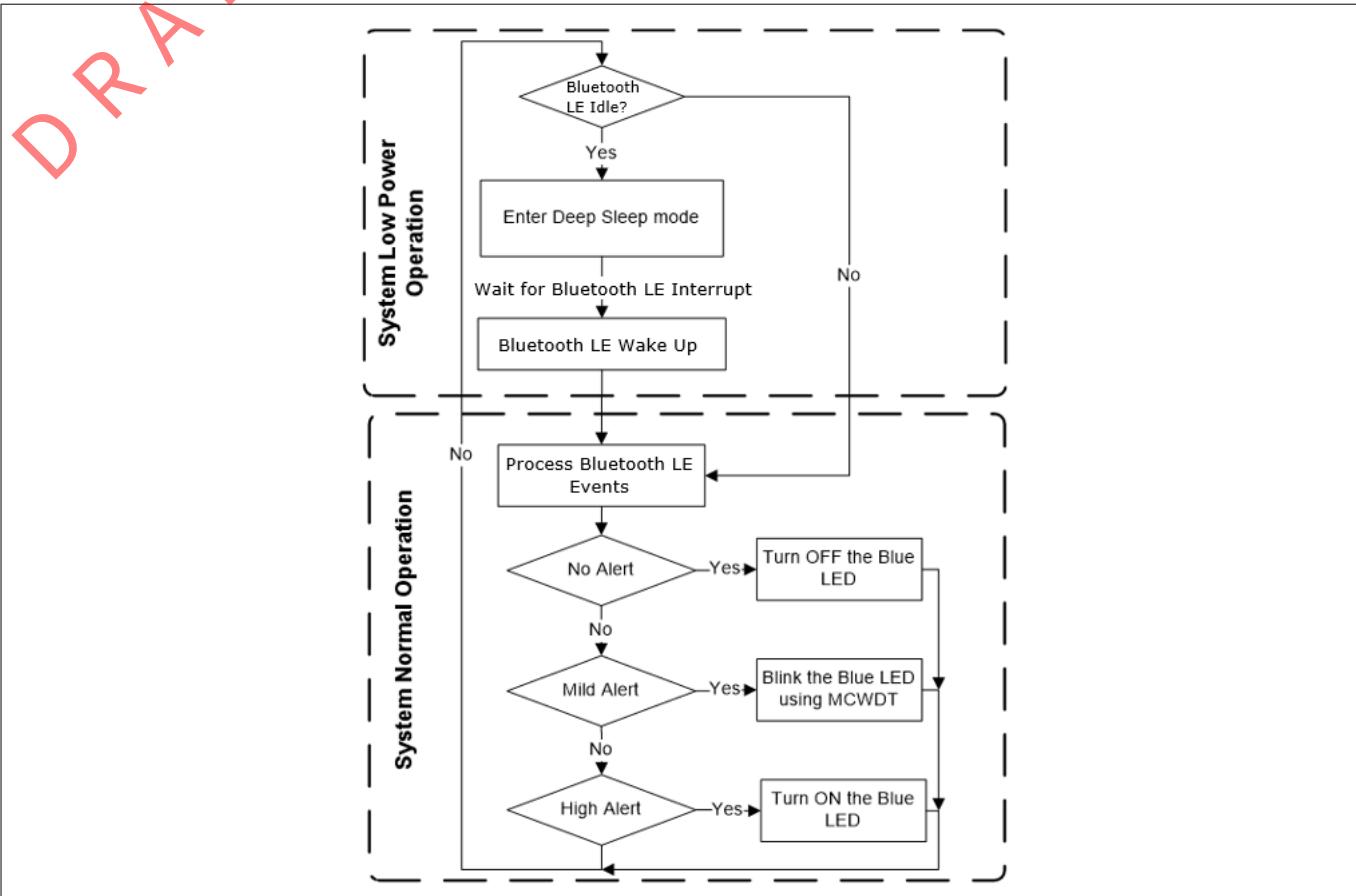


Figure 108 Firmware main loop flowchart

5 PSoC™ 6 application notes

If there are no pending Bluetooth® LE host events and there is no active interaction, the application goes into low-power mode. It then tells the Bluetooth® LE to process events, and updates the LEDs based on the alert level.

```

/* The call to EnterLowPowerMode also causes the device to enter hibernate mode if the
BLE is disconnected. */

EnterLowPowerMode();
/* Cy_Ble_ProcessEvents() allows BLE stack to process pending events */
Cy_BLE_ProcessEvents();
/* Update Alert Level value on the Blue LED */
switch(alertLevel)
{
    case CY_BLE_NO_ALERT:
        /* Disable MCWDT interrupt at NVIC */
        NVIC_DisableIRQ(MCWDT_isr_cfg.intrSrc);
        /* Turn the Blue LED OFF in case of no alert */
        Cy_GPIO_Write(Alert_LED_0, LED_OFF);
        break;

    /* Use the MCWDT to blink the Blue LED in case of mild alert */
    case CY_BLE_MILD_ALERT:
        /* Enable MCWDT interrupt at NVIC */
        NVIC_EnableIRQ(MCWDT_isr_cfg.intrSrc);
        /* The MCWDT interrupt handler will take care of LED blinking */
        break;

    case CY_BLE_HIGH_ALERT:
        /* Disable MCWDT interrupt at NVIC */
        NVIC_DisableIRQ(MCWDT_isr_cfg.intrSrc);
        /* Turn the Blue LED ON in case of high alert */
        Cy_GPIO_Write(Alert_LED_0, LED_ON);
        break;

    /* Do nothing in all other cases */
    default:
        break;
}

```

This completes the summary of how the firmware works in the code example. Feel free to explore the source files for a deeper understanding.

5.3.5.8 Part 5. Build the project, program the Device

This section shows how to program the PSoC™ 6-BLE device. If you are using a development kit with a built-in programmer (the [CY8CKIT-062-BLE PSoC™ 6-BLE Pioneer Kit](#), for example), connect the board to your computer using the USB cable. If you are developing on your own hardware, you may need a hardware programmer/debugger; for example, a [CY8CKIT-002 MiniProg3](#).

5 PSoC™ 6 application notes

Path	Working from scratch code example as reference only	Using code example new to PSoC™ Creator or Bluetooth® LE	Using code example familiar with PSoC™ Creator and Bluetooth® LE
Actions	Perform all steps		

If you are working from scratch and encounter errors, revisit prior steps to ensure that you accomplished all the required tasks. You can work to resolve errors or switch to the code example for these final steps.

Note: CY8CKIT-062-BLE kit ships with Kitprog2. Before programming the kit, ensure that the board is updated with latest Kitprog3 firmware. Refer [Debugging](#) section on how to update the Kitprog firmware.

- **Select the debug target**

PSoC™ Creator can debug one core at a time.

- In PSoC™ Creator, choose **Debug > Select Debug Target**, as Figure 109 shows

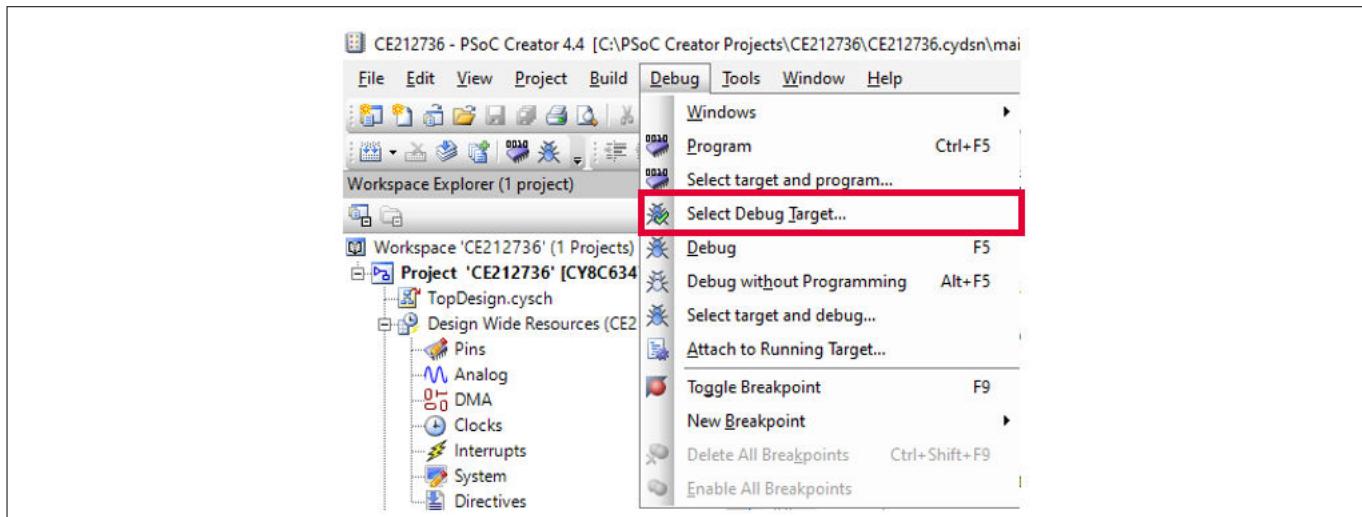


Figure 109 Selecting debug target

- **Connect to the board**

In the **Select Debug Target** dialog box, select the CM4 target, then click **OK** or **Connect**, as Figure 110 shows.

5 PSoC™ 6 application notes

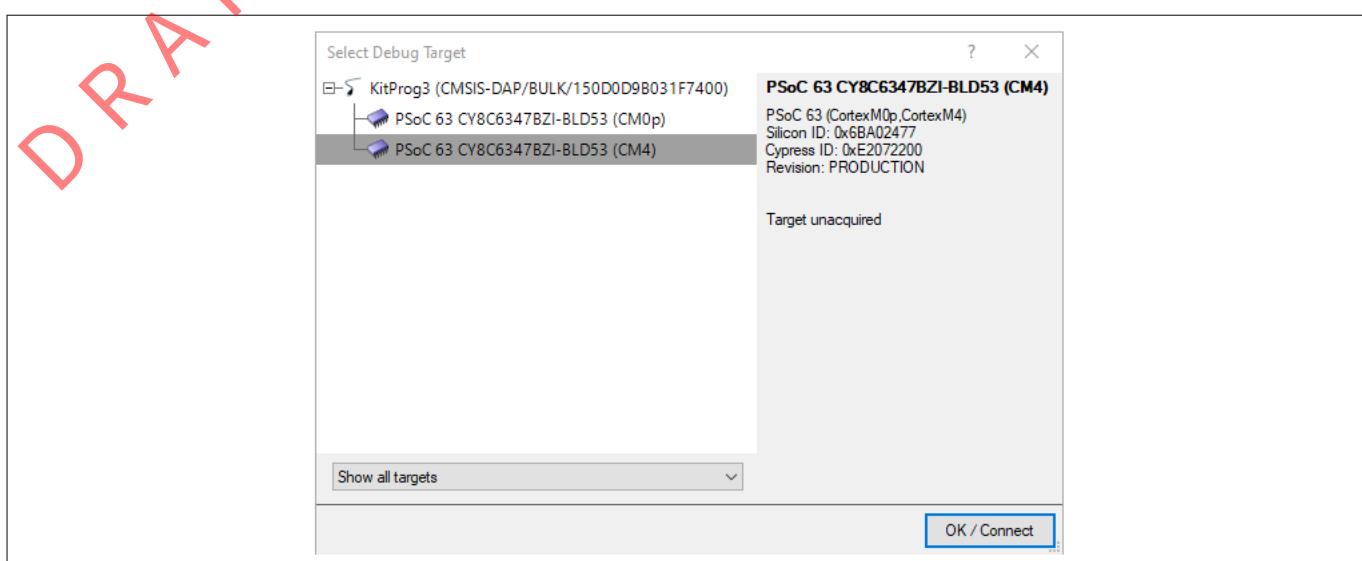


Figure 110 Connecting to a device

Tip: For programming the board you can pick either target. The cores share the same memory space. Programming either core programs both cores. However, if you are debugging this choice matters. The debugger will see only the core you connect to. These instructions do not use the debugger.

- Program the board

Choose **Debug > Program** to program the device with the project, as [Figure 111](#) shows.

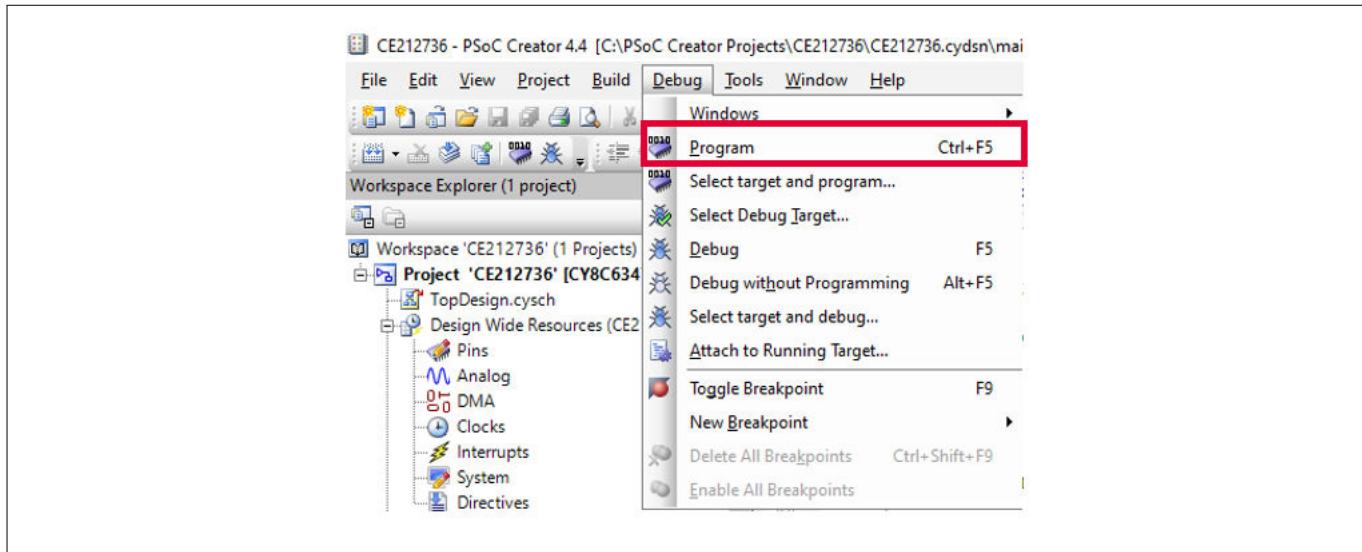


Figure 111 Programming the device

You can view the programming status in the lower left corner of the window PSoC™ Creator window, as [Figure 112](#) shows.

5 PSoC™ 6 application notes

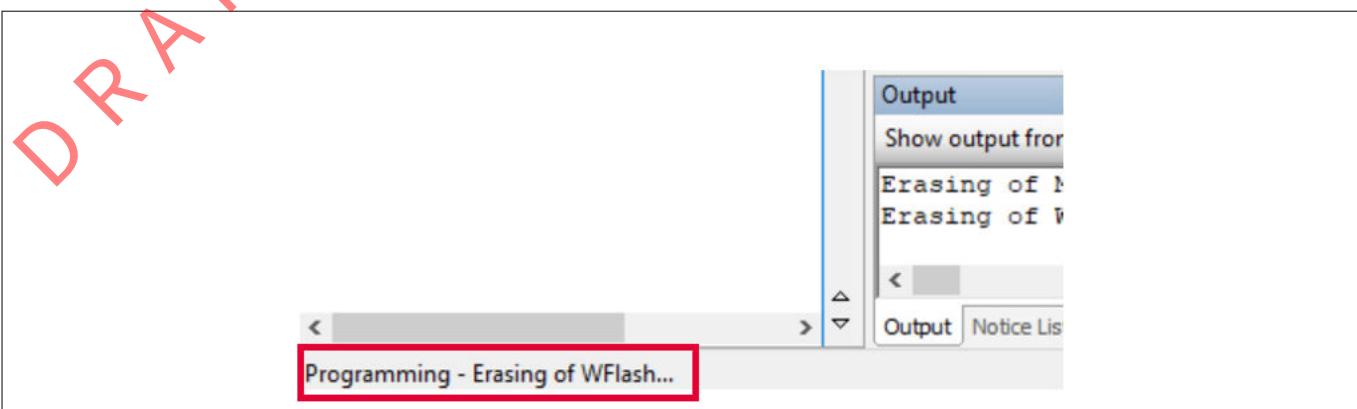


Figure 112 Programming status

In a dual core application, the linker files put each executable at the correct location in memory. Execution begins on the CM0+ core, which enables the CM4 core.

When programming is complete, the application runs. The LED turns green, indicating that the target is advertising. After the advertising timeout occurs it turns red, indicating that it is disconnected.

Tip: *The **Debug > Debug** command also programs the board. If any code needs to be generated or rebuilt, that happens automatically when you issue a **Program** or **Debug** command. You can also debug without programming the board. However, these instructions do not use the debugger.*

Note: *The KitProg2 firmware on the kit might require an update. Please refer to the kit user guide for step-by-step instructions on updating the firmware.*

5.3.5.9 Part 6. Test your design

This section describes how to test your Bluetooth® LE design using either the [Appendix D.3](#) or the [Appendix D.2](#). The setup for testing your design using the Bluetooth® LE Pioneer Kit is shown in [Figure 71](#).

Path	Working from scratch code example as reference only	Using code example new to PSoC™ Creator or Bluetooth® LE	Using code example familiar with PSoC™ Creator and Bluetooth® LE
Actions	Perform either step 1 or step 2		

- **Test using the CySmart Mobile App**
 - Turn on Bluetooth® on your iOS or Android device
 - Launch the CySmart app
 - Press the reset switch on the Bluetooth® LE Pioneer Kit to start Bluetooth® LE advertisements from your design. The green LED must be on for the CySmart app to see the device
 - Pull down the CySmart app home screen to start scanning for BLE Peripherals. The Find Me target appears as a Bluetooth® LE device in the CySmart app home screen. Tap to establish a Bluetooth® LE connection. If the phone does not find the target, press the reset button again, or try the CySmart Host Emulation tool (step 2)
 - Select the “Find Me” Profile from the carousel view
 - Select an **Alert Level** value on the **Find Me** Profile screen and watch the state of the blue LED on your device change per your selection

[Figure 113](#) shows this process using the iOS app. [Figure 114](#) shows the process using the Android app.

5 PSoC™ 6 application notes

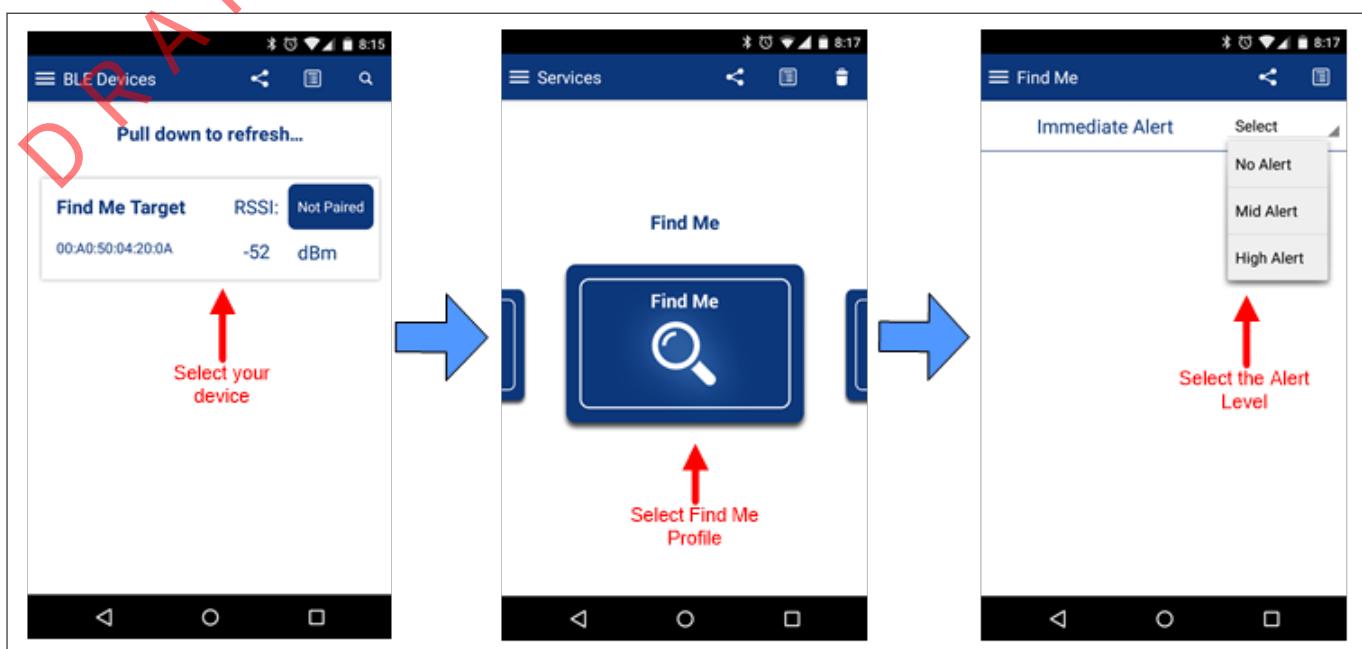


Figure 113 Testing with CySmart iOS App

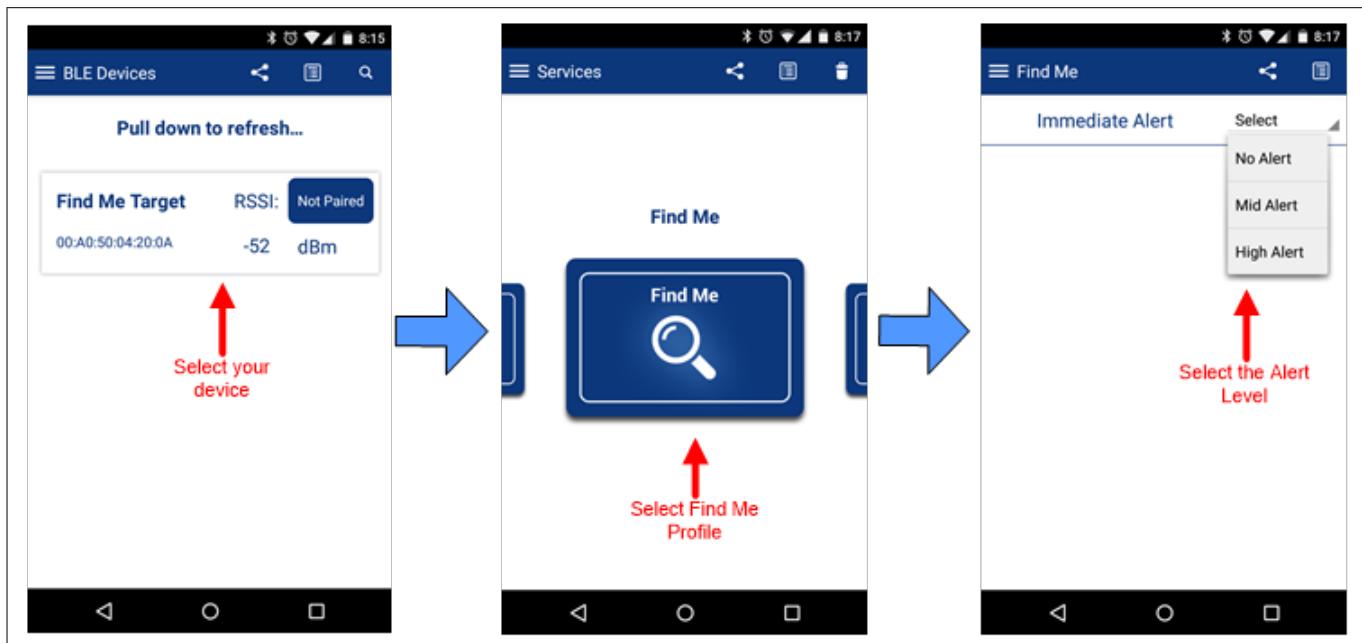


Figure 114 Testing with CySmart Android App

- **Test using the CySmart Host Emulation Tool**

As an alternative to the CySmart mobile app, you can use the CySmart Host Emulation Tool to establish a Bluetooth® LE connection with your design and perform read or write operations on Bluetooth® LE characteristics.

- If not already installed, install the CY Smart Host Emulation Tool

Right-click the Bluetooth® Low Energy Component symbol on the Top Schematic and select **Download CySmart** as shown in [Figure 115](#). Follow the installer directions to install the tool.

5 PSoC™ 6 application notes

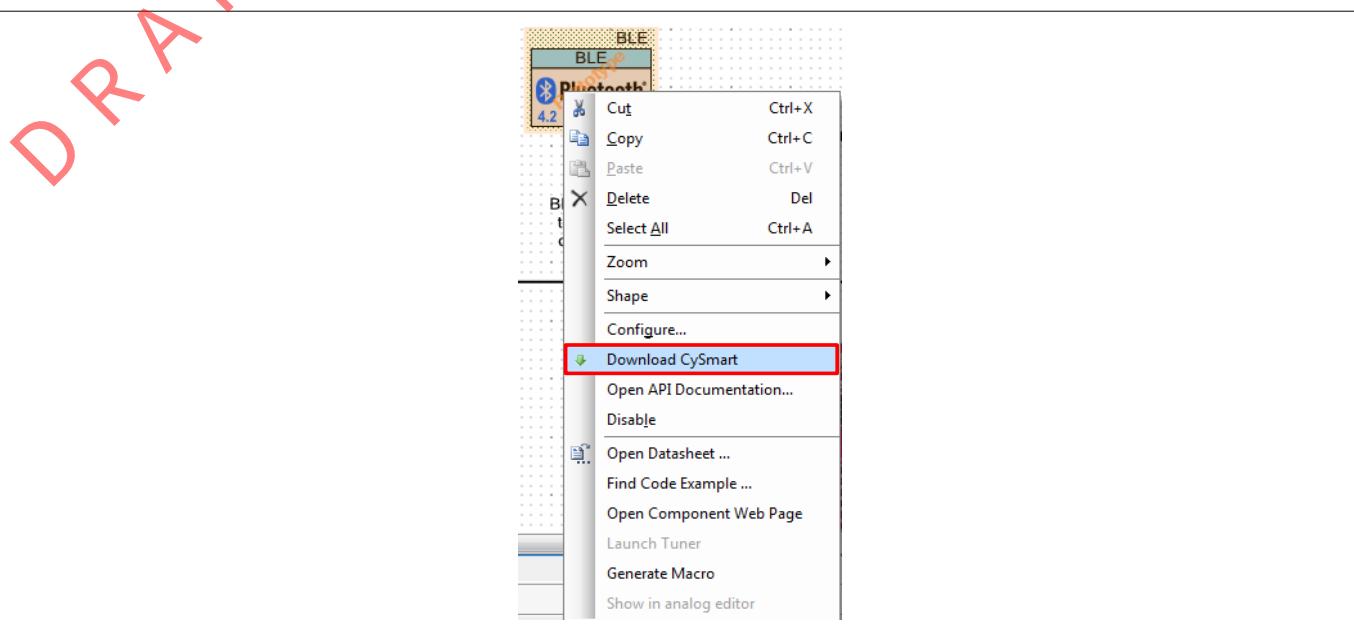


Figure 115 Download CySmart

- Connect the Bluetooth® LE dongle to your Windows personal computer. Wait for the driver installation to be completed
- Launch the CySmart Host Emulation Tool

The tool automatically detects the Bluetooth® LE dongle. Click **Refresh** if the Bluetooth® LE dongle does not appear in the **Select BLE Dongle Target** pop-up window. Click **Connect**, as shown in [Figure 116](#).

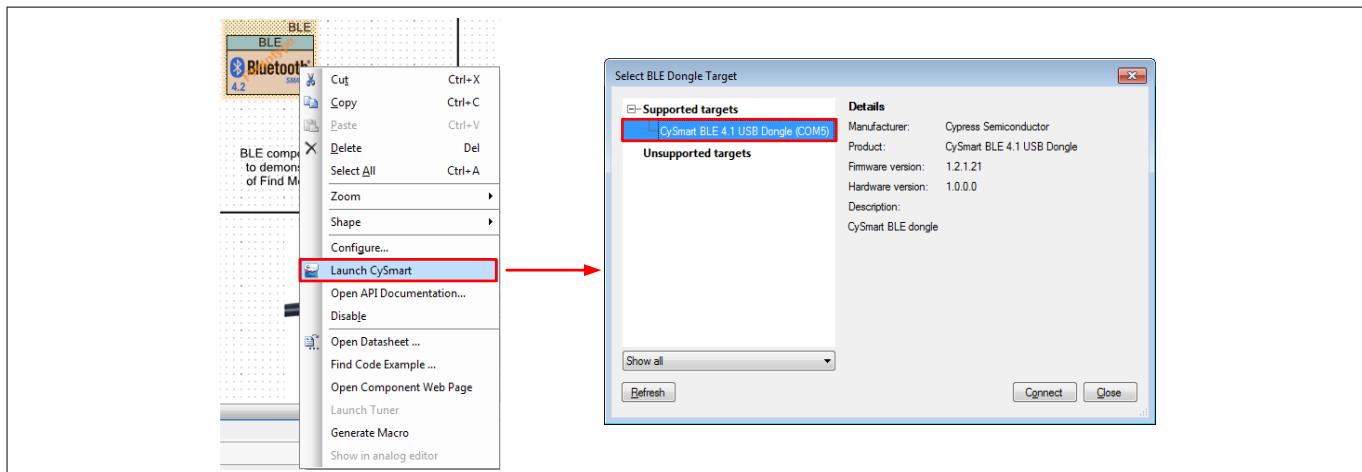


Figure 116 CySmart Bluetooth® LE dongle selection

Note: If the dongle firmware is outdated, you will be alerted. You must upgrade the firmware before you can complete this step. Follow the instructions in the window to update the dongle firmware.

- Click **Configure Master Settings** and then click **Restore Defaults**, as shown in [Figure 117](#). Then, click **OK**

5 PSoC™ 6 application notes

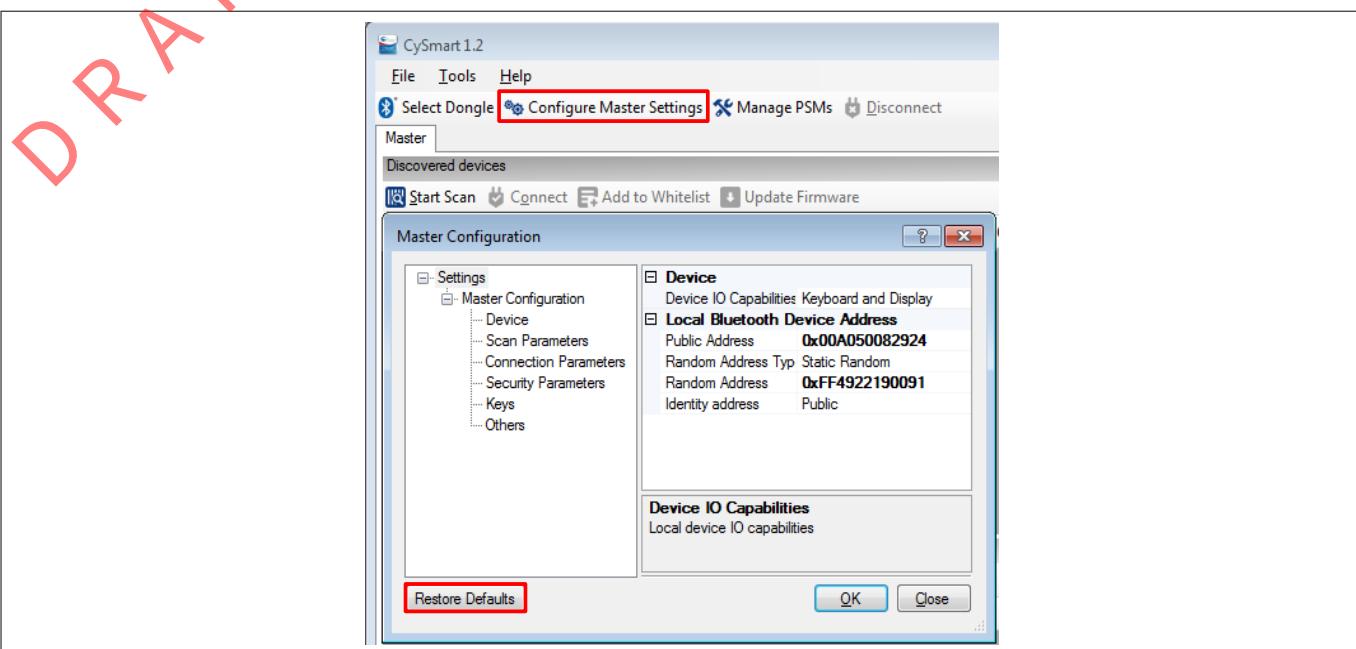


Figure 117 CySmart master settings configuration

- Press the reset switch on the Bluetooth® LE pioneer kit to start BLE advertisements from your design. The LED turns green to indicate you are advertising
- On the CySmart Host Emulation Tool, click **Start Scan**. Your device name (configured as **Find Me Target**) should appear in the **Discovered devices** list, as shown in [Figure 118](#)

Note: If the scan process times out without finding the target, the LED turns red. Click the reset button again to resume advertising.

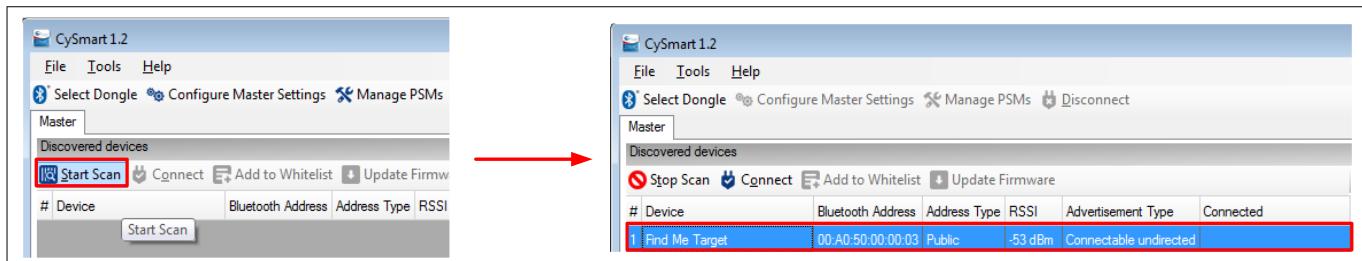


Figure 118 CySmart device discovery

- Select **Find Me Target** and click **Connect** to establish a Bluetooth® LE connection between the CySmart Host Emulation Tool and your device, as shown in [Figure 119](#)

5 PSoC™ 6 application notes

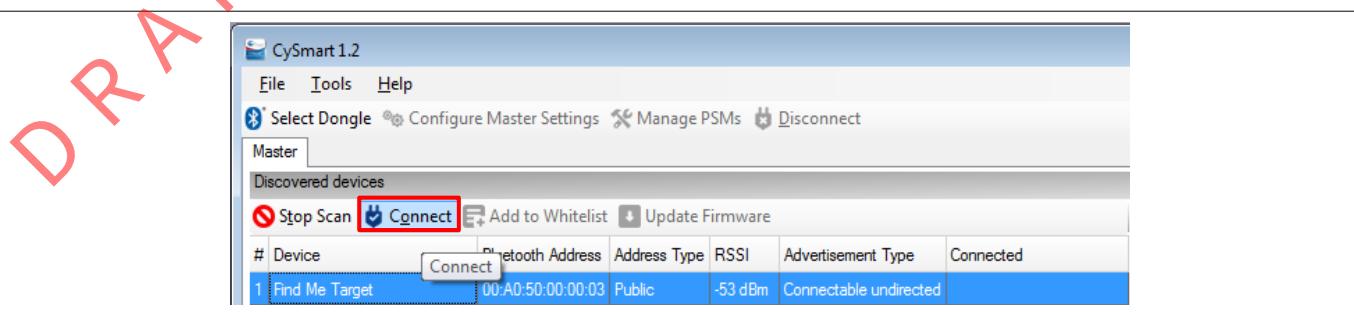


Figure 119 CySmart device connection

- Once connected, switch to the **Find Me Target** device tab and discover all the attributes on your design from the CySmart Host Emulation Tool, as shown in [Figure 120](#)

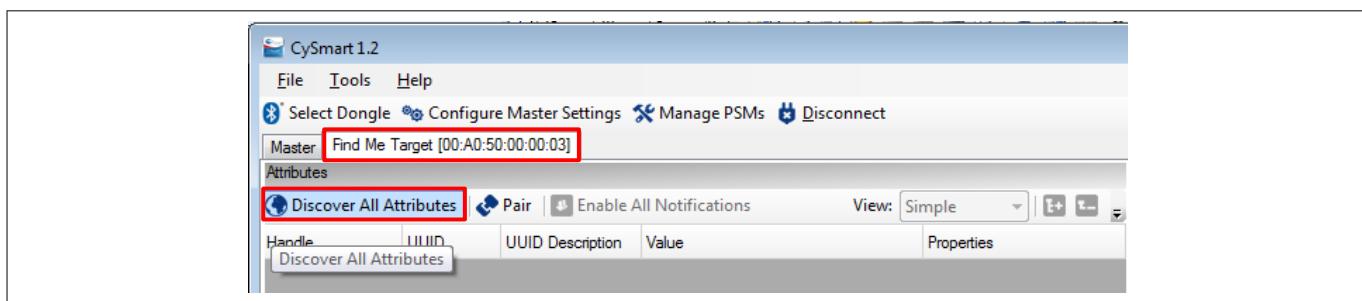


Figure 120 CySmart attribute discovery

- Scroll down the **Attributes** window and locate the **Immediate Alert** Service fields. Write a value of 0, 1, or 2 to the **Alert Level** characteristic under the **Immediate Alert** Service, as [Figure 121](#) shows. Observe the state of the LED on your device change per your Alert level characteristic configuration

5 PSoC™ 6 application notes

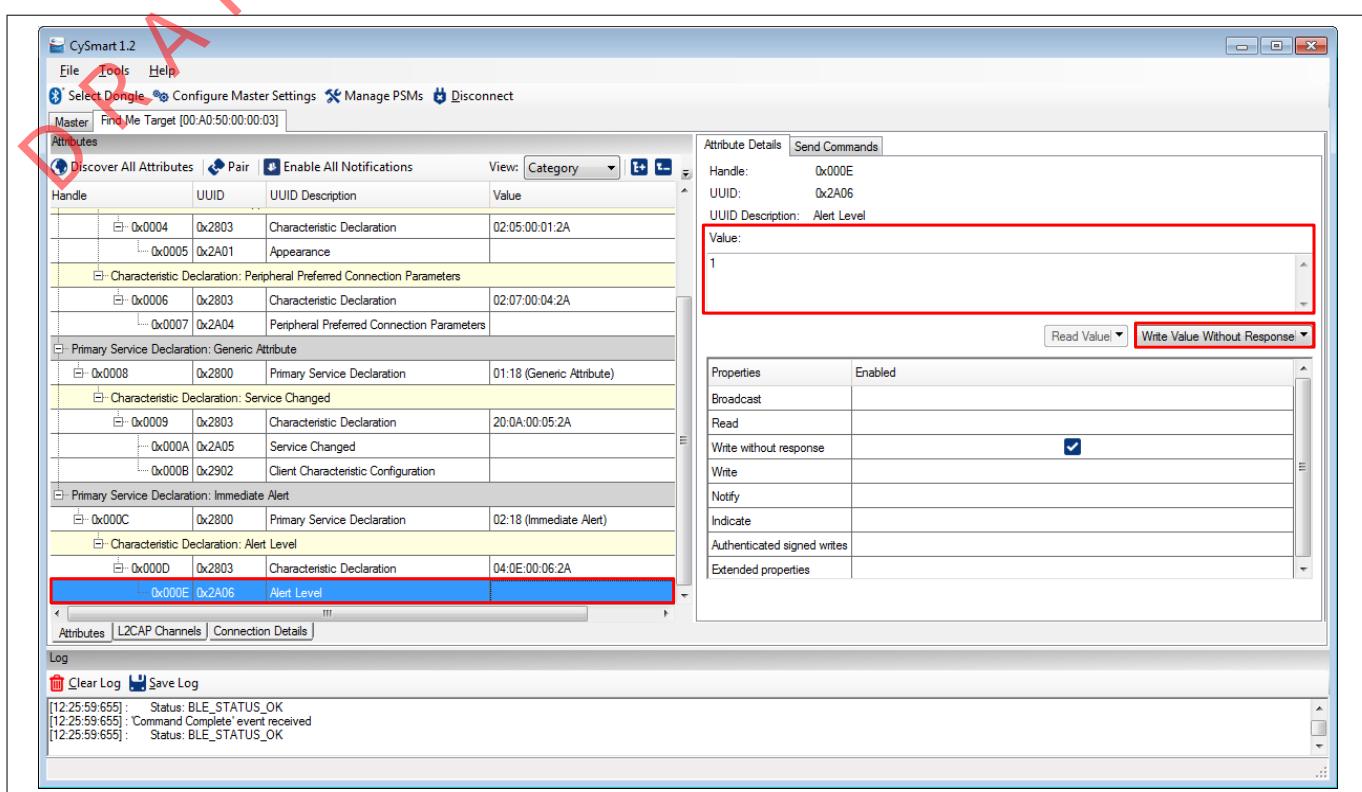


Figure 121 Testing with CySmart Host Emulation Tool

- When you are finished exploring, click the Disconnect button in the CySmart application. The LED turns red to indicate you are disconnected

You can repeat this process by pressing the reset button on the board to resume advertising.

5 PSoC™ 6 application notes**5.3.6 Summary**

This application note explored the PSoC™ 6-BLE device architecture and the associated development tools. PSoC™ 6-BLE is a truly programmable embedded system-on-chip, integrating low-power Bluetooth® LE radio, configurable analog and digital peripheral functions, memory, and a dual-CPU system on a single chip. The integrated features and low-power modes make PSoC™ 6-BLE an ideal choice for battery-operated wearable, health, and fitness Bluetooth® LE applications.

~~5 PSoC™ 6 application notes~~

~~5.3.7 Related application notes and code examples~~

For a complete and updated list of PSoC™ 6 MCU code examples, please visit our [code examples webpage](#). For more PSoC™ 6 MCU related documents, visit our [PSoC™ 6 MCU product webpage](#).

[Table 10](#) lists the system-level and general application notes that are recommended for the next steps in learning about PSoC™ 6-BLE and PSoC™ Creator.

Table 10 General and system-level application notes, code examples

Document	Document name
AN221774	Getting Started with PSoC™ 6 MCU on PSoC™ Creator
CE221773	PSoC™ 6 MCU Hello World Example
AN218241	PSoC™ 6 MCU Hardware Design Considerations
AN219434	PSoC™ 6 MCU Importing Generated Code into an IDE
AN219528	PSoC™ 6 MCU Low-Power Modes and Power Reduction Techniques

[Table 11](#) lists the application notes and code examples (CE) for specific peripherals and applications of the device.

Table 11 Documents related to PSoC™ 6-BLE features

Document	Document name
Programmable digital	
Bluetooth® smart	
AN91162	Creating a BLE Custom Profile
AN91445	Antenna Design and RF Layout Guidelines
AN92584	Designing for Low Power and Estimating Battery Life for BLE applications
CE218463	Bluetooth® Low Energy (BLE) Alert Notification Client/Server
CE218464	Bluetooth® Low Energy (BLE) Phone Alert Client/Server
System resources, CPU, and interrupts	
AN215656	PSoC™ 6 MCU Dual-CPU System Design
AN217666	PSoC™ 6 MCU Interrupts
CE216795	PSoC™ 6 MCU Dual-CPU Basics
CE216825	PSoC™ 6 MCU Real-Time Clock Basics
CE218129	PSoC™ 6 MCU Wake up from Hibernate Using Low-Power Comparator
CE218541	PSoC™ 6 MCU Fault-Handling Basics
CE218542	PSoC™ 6 Custom Tick Timer Using RTC Alarm Interrupt
CE218552	PSoC™ 6 MCU UART to Memory Buffer Using DMA
CE218964	PSoC™ 6 MCU RTC Daily Alarm
CE219339	PSoC™ 6 MCU MCWDT and RTC Interrupts (Dual Core)
CE219521	PSoC™ 6 MCU GPIO Interrupt

(table continues...)

~~5 PSoC™ 6 application notes~~

Table 11 (continued) Documents related to PSoC™ 6-BLE features

Document	Document name
CE219881	PSoC™ 6 MCU Switching Power Modes
CE220060	PSoC™ 6 MCU Watchdog Timer
CE220061	PSoC™ 6 MCU Multi-Counter Watchdog Interrupts
CE220120	PSoC™ 6 MCU Blocking Mode Flash Write
CE220169	PSoC™ 6 MCU Periodic Interrupt Using TCPWM
GPIO	
CE219490	PSoC™ 6 Breathing LED Using SMART IO
CE219506	PSoC™ 6 Clock Buffer Using SMART IO
CE220263	PSoC™ 6 MCU GPIO Pins Example
CAPSENSE™	
AN92239	Proximity Sensing with CAPSENSE™
AN85951	PSoC™ 4 and PSoC™ 6 MCU CAPSENSE™ Design Guide
SMIF	
Bootloader	
AN213924	MCU Bootloader Software Development Kit (SDK) Guide
CE213903	PSoC™ 6 MCU Basic Bootloaders
Communications	
CE220541	PSoC™ 6 MCU SCB EzI2C
Segment LCD	
Audio	
CE218636	PSoC™ 6 MCU Inter-IC Sound (I2S) Example
CE219431	PSoC™ 6 MCU PDM-to-PCM Example
RTOS	
CE217911	PSoC™ 6 FreeRTOS™ Example Project
Security	
CE220465	PSoC™ 6 MCU Cryptography – AES Demonstration
CE220511	PSoC™ 6 MCU Cryptography – SHA Demonstration

5 PSoC™ 6 application notes

A Appendix A. Glossary

This section lists the most commonly used terms that you might encounter while working with PSoC™ family of devices.

Component Customizer: Simple GUI in PSoC™ Creator that is embedded in each Component. It is used to customize the Component parameters and is accessed by right-clicking a Component.

Components: Components are used to integrate multiple ICs and system interfaces into one PSoC™ Component that is inherently connected to the MCU via the main system bus. For example, the Bluetooth® Low Energy component creates Bluetooth® Smart products in minutes. Similarly, you can use the Programmable Analog components for sensors.

MiniProg3: Programming hardware for development that is used to program PSoC™ devices on your custom board or PSoC™ development kits that do not support a built-in programmer.

PSoC™: A programmable, embedded design platform that includes one or more CPUs, such as the 32-bit CM4, with both analog and digital programmable blocks. It accelerates embedded system design with reliable, easy-to-use solutions, such as touch sensing, and enables low-power designs.

PSoC™ Creator: PSoC™ 3, PSoC™ 4, PSoC™ 5LP, and PSoC™ 6-BLE IDE software that installs on your PC and allows concurrent hardware and firmware design of PSoC™ systems, or hardware design followed by export to other popular IDEs.

Peripheral Driver Library: The Peripheral Driver Library (PDL) simplifies software development for the PSoC™ 6 MCU architecture. The PDL reduces the need to understand register usage and bit structures, thus easing software development for the extensive set of peripherals available.

PSoC™ Programmer: A flexible, integrated programming application for programming PSoC™ devices. PSoC™ Programmer is integrated with PSoC™ Creator to program PSoC™ 3, PSoC™ 4, PRoC, PSoC™ 5LP, and PSoC™ 6 MCU designs.

5 PSoC™ 6 application notes

B Appendix B. Bluetooth® LE protocol

B.1 Overview

BLE, also known as Bluetooth® Smart, was introduced by the Bluetooth® SIG as a low-power wireless standard operating in the 2.4-GHz ISM band. [Figure 122](#) shows the BLE protocol stack.

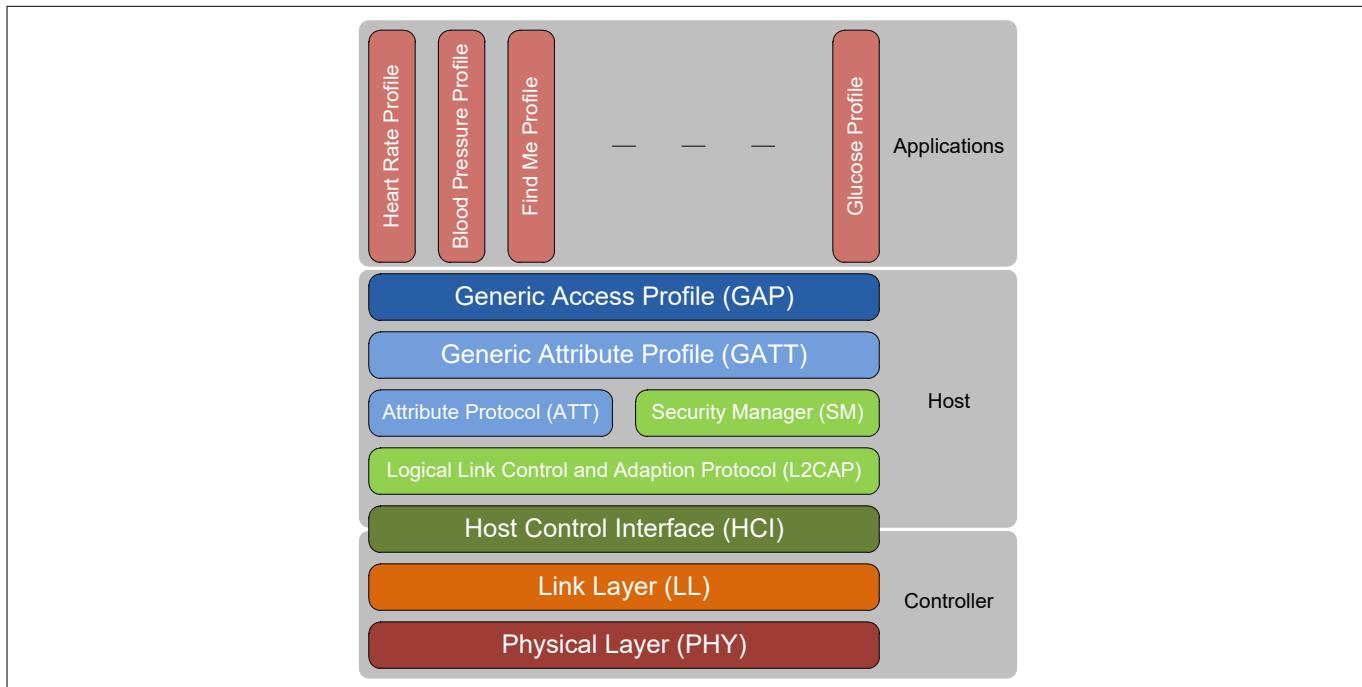


Figure 122 **Bluetooth® LE architecture**

The BLE stack can be subdivided into three groups:

- **Controller:** A physical device that encodes the packet and transmits it as radio signals. On reception, the controller decodes the radio signals and reconstructs the packet
- **Host:** A software stack consisting of various protocols and Profiles (Security Manager, Attribute Protocol, and so on) that manages how two or more devices communicate with one another
- **Application:** A use case that uses the software stack and the controller to implement a particular functionality

The following sections provide an overview of the multiple layers of the Bluetooth® LE stack, using the standard Heart Rate and Battery Service as examples. For a detailed Bluetooth® LE architecture description, see the [Bluetooth® Core Specification](#).

B.2 Physical Layer (PHY)

The physical layer transmits or receives digital data at 1 Mbps using Gaussian frequency-shift keying (GFSK) modulation in the 2.4-GHz ISM band. The Bluetooth® LE physical layer divides the ISM band into 40 RF channels with a channel spacing of 2 MHz, 37 of which are data channels and 3 are advertisement channels.

B.3 Link Layer (LL)

The link layer implements key procedures to establish a reliable physical link (using an acknowledgement and flow-control-based architecture) and features that help make the Bluetooth® LE protocol robust and low-power. Some link layer functions include:

5 PSoC™ 6 application notes

- Advertising, scanning, creating, and maintaining connections to establish a physical link
- 24-bit CRC and AES-128-bit encryption for robust and secure data exchange
- Establishing fast connections and low-duty-cycle advertising for low-power operation
- Adaptive Frequency Hopping (AFH), which changes the communication channel used for packet transmission so that the interference from other devices is reduced

At the link layer, two roles are defined:

- Master:** A smartphone is an example that configures the link layer in the master configuration
- Slave:** A heart-rate monitor device is an example that configures the link layer in the slave configuration

PSoC™ 6-BLE devices can operate in either configuration.

The link-layer slave is the one that advertises its presence to another link-layer master. A link-layer master receives the advertisement packets and can choose to connect to the slave based on the request from an application (see [Figure 123](#)). In this example implementation of a heart-rate monitor application, a heart-rate monitor device acts as the slave and sends the data to a smartphone, which acts as the master. A smartphone app then can display the reading on the smartphone.

PSoC™ 6-BLE devices implement the time-critical and processor-intensive parts of the link layer such as advertising, CRC, and AES encryption in hardware. Link-layer control operations such as entering the advertisement state and starting encryption are implemented in firmware.

[Figure 123](#) shows the Bluetooth® LE link-layer packet structure and sizes of the individual fields in the link-layer packet. The link-layer packet carries all upper layer data in its payload field. It has a 4-byte access address that is used to uniquely identify communications on a physical link, and ignore packets from a nearby Bluetooth® LE device operating in the same RF channel. 24-bit CRC provides data robustness.

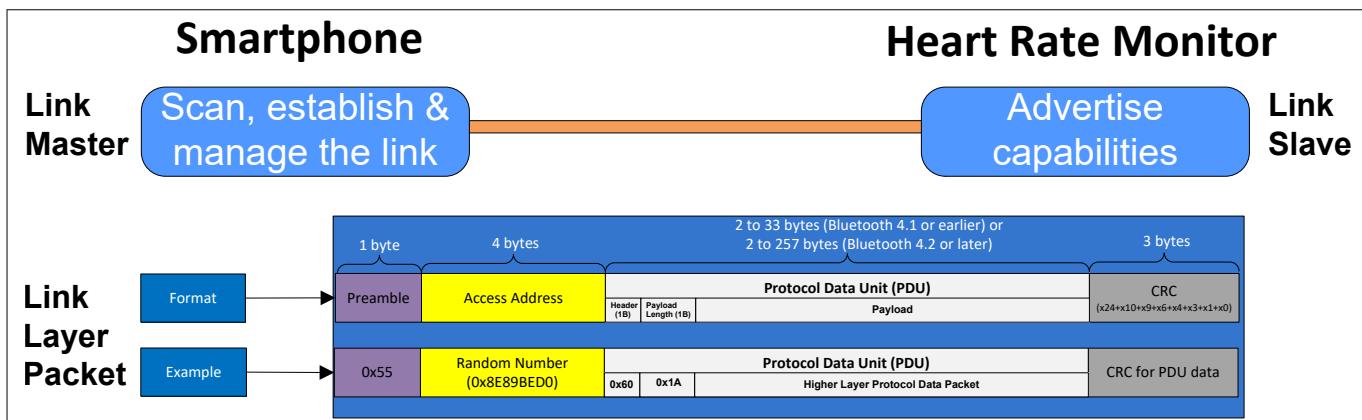


Figure 123 **Bluetooth® LE link layer protocol**

B.4 Host Control Interface (HCI)

The HCI is the standard-defined interface between the host and the controller. It allows the host and the controller to exchange information such as commands, data, and events over different physical transports such as USB or UART. The HCI requires a physical transport only when the controller and the host are different devices.

In PSoC™ 6-BLE devices, the HCI is just a firmware protocol layer that passes the messages and events between the controller and the host.

B.5 Logical Link Control and Adaptation protocol (L2CAP)

L2CAP provides protocol multiplexing, segmentation, and reassembly services to upper-layer protocols. Segmentation breaks the packet received from the upper layer into smaller packets that the link layer can transmit, while reassembly combines the smaller packets received from the link layer into a meaningful packet.

~~5 PSoC™ 6 application notes~~

The L2CAP layer supports three protocol channel IDs for [Appendix B.7](#), and L2CAP control, as shown in [Figure 124](#). Bluetooth 4.2 allows direct data channels through the L2CAP connection-oriented channels on top of these protocol channels.

The L2CAP and the layers above it are implemented in firmware in PSoC™ 6-BLE.

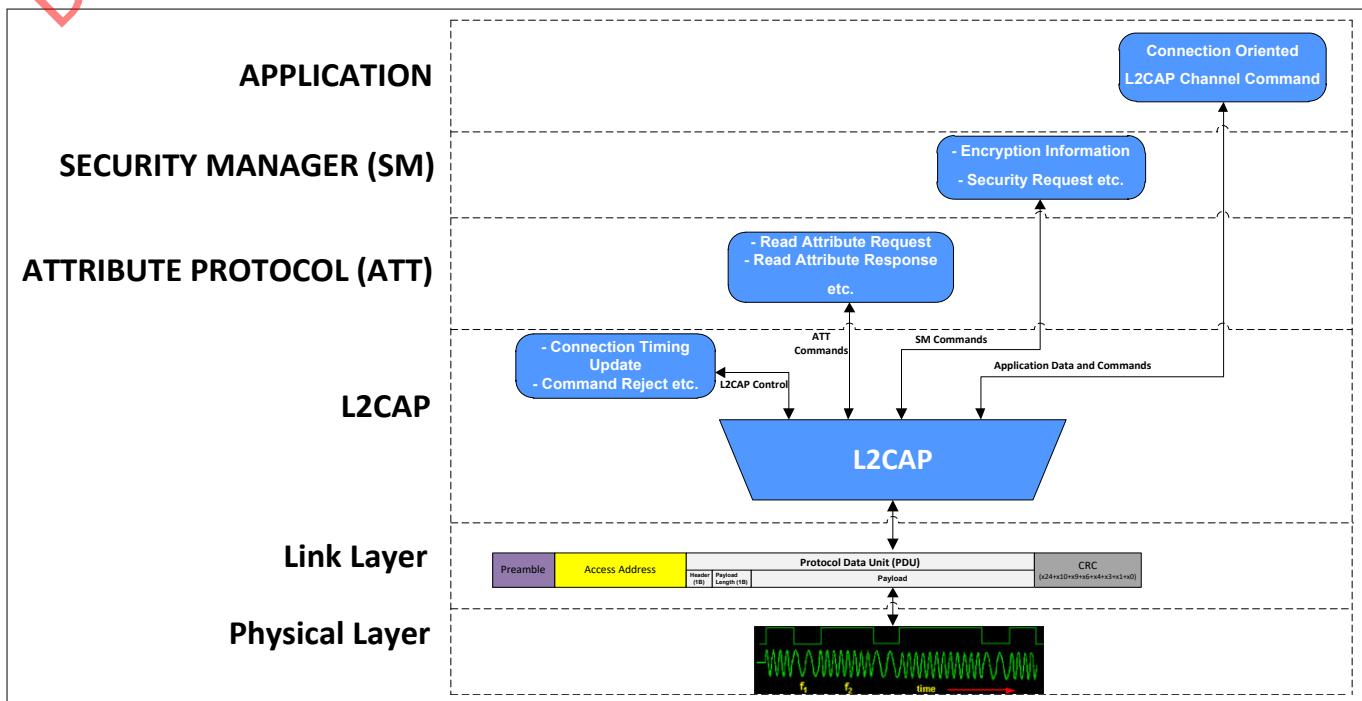


Figure 124 **Bluetooth® LE L2CAP layer**

B.6 Security manager (SM)

The SM layer defines the methods used for pairing, encryption, and key distribution.

- **Pairing** is the process to enable security features. In this process, two devices are authenticated, the link is encrypted, and then the encryption keys are exchanged. This enables the secure exchange of data over the Bluetooth® LE interface without being snooped on by a silent listener on the RF channel
- **Bonding** is the process in which the keys and the identity information exchanged during the pairing process are saved. After devices are bonded, they do not have to go through the pairing process again when reconnected

Bluetooth® LE uses 128-bit [AES](#) for data encryption.

B.7 Attribute protocol (ATT)

There are two GATT roles in BLE that you should know to understand the ATT and GATT layers:

- **GATT Server:** A GATT Server contains the data or information. It receives requests from a GATT Client and responds with data. For example, a heart-rate monitor GATT Server contains heart-rate information; a BLE HID keyboard GATT Server contains user key press information
- **GATT Client:** A GATT Client requests and/or receives data from a GATT Server. For example, a smartphone is a GATT Client that receives heart-rate information from the heart-rate GATT Server; a laptop is a GATT Client that receives key-press information from a Bluetooth® LE keyboard

ATT forms the basis of Bluetooth® LE communication. This protocol enables the GATT Client to find and access data or attributes on the GATT Server. For more details about the GATT Client and Server architecture, see [Appendix B.8](#).

5 PSoC™ 6 application notes

An attribute is the fundamental data container in the ATT/GATT layer, which consists of the following:

- **Attribute handle:** The 16-bit address used to address and access an attribute
- **Attribute type:** This specifies the type of data stored in an attribute. It is represented by a 16-bit UUID defined by the Bluetooth® SIG

For example, the 16-bit UUID of the Heart-Rate Service is 0x180D; the UUID for the Device Name Attribute is 0x2A00. Visit the Bluetooth® [webpage](#) for a list of 16-bit UUIDs assigned by the SIG.

- **Attribute value:** This is the actual data stored in the attribute
- **Attribute permission:** This specifies the attribute access, authentication, and authorization requirements. Attribute permission is set by the higher layer specification and is not discoverable through the attribute protocol

Figure 125 shows the structure of a Device name attribute as an example.

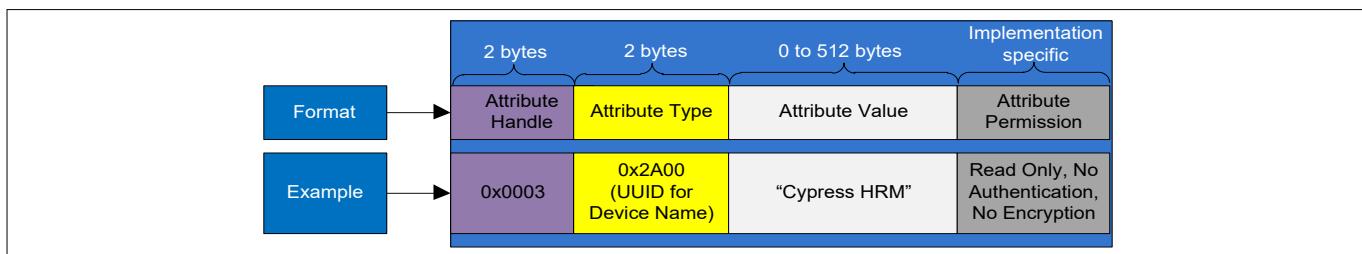


Figure 125 Attribute format example

B.7.1 Attribute hierarchy

Attributes are the building blocks for representing data in ATT/GATT. Attributes can be broadly classified into the following two groups to provide hierarchy and abstraction of data:

- **Characteristic:** A collection of attributes that exposes the system information or meaningful data. A Characteristic consists of the following attributes:
 - Characteristic declaration attribute: This defines the beginning of a characteristic
 - Characteristic value attribute: This holds the actual data
 - Characteristic descriptor attributes: These are optional attributes, which provide additional information about the characteristic value

“Battery Level” is an example of a characteristic in the Battery Service (BAS). Representing the battery level in percentage values is an example of a characteristic descriptor.

Figure 126 shows the structure of a characteristic with Battery Level as an example.

- The first part of a characteristic is the declaration of the characteristic (it marks the beginning of a characteristic) indicated by the Battery Level Characteristic in Figure 126
- Next is the actual characteristic value or the real data, which in the case of the Battery Level Characteristic is the current battery level. The battery level is expressed as a percentage of full scale, for example “65,” “90,” and so on
- Characteristic descriptors provide additional information that is required to make sense of the characteristic value. For example, the Characteristic Presentation Format Descriptor for Battery Level indicates that the battery level is expressed as a percentage. Therefore, when “90” is read, the GATT Client knows this is 90 percent and not 90 mV or 90 mAh. Similarly, the Valid Range Characteristic descriptor (not shown in Figure 126) indicates that the battery level range is between 0 and 100 percent
- A Client Characteristic Configuration Descriptor (CCCD) is another commonly used Characteristic descriptor that allows a GATT Client to configure the behavior of a Characteristic on the GATT Server. When the GATT Client writes a value of 0x01 to the CCCD of a Characteristic, it enables asynchronous notifications (described in the next section) to be sent from the GATT Server. In the case of a Battery Level Characteristic,

5 PSoC™ 6 application notes

writing 0x01 to the Battery Level CCCD enables the Battery Service to notify its battery level periodically or on any change in battery-level value

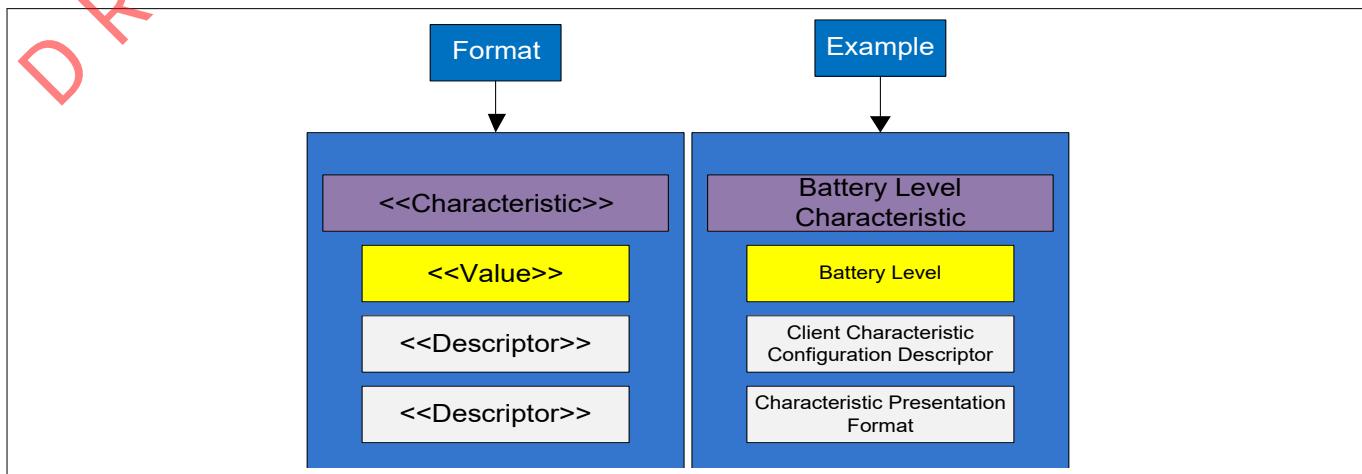


Figure 126 Characteristic format and example

- **Service:** The type of attribute that defines a function performed by the GATT Server. A service is a collection of characteristics and can include other services. The concept of a service is used to establish the grouping of relative data and provide a data hierarchy. See [Figure 127](#) for an example of a Heart Rate Service ([HRS](#)).

A service can be of two types: A primary service or a secondary service. A primary service exposes the main functionality of the device, while the secondary service provides additional functionality. For example, in a heart-rate monitoring device, the HRS is a primary service and BAS is a secondary service.

A service can also include other services that are present on the GATT Server. The entire included services become part of the new service.

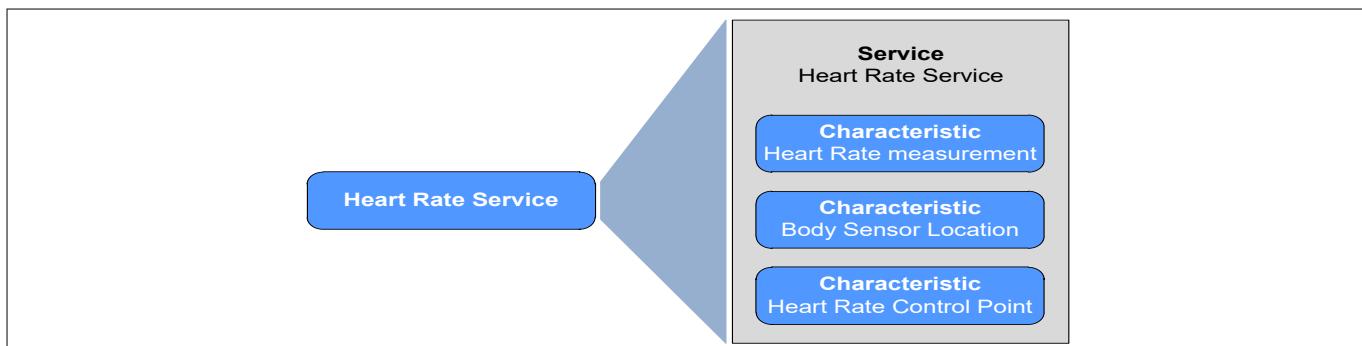


Figure 127 Bluetooth® LE Heart rate service example

The word “Profile” in Bluetooth® LE is a collection of services and their behavior that together perform a particular end application. A Heart Rate Profile ([HRP](#)) is an example of a Bluetooth® LE Profile that defines all the required services for creating a heart-rate monitoring device. See the [Appendix B.9](#) section for details.

[Figure 128](#) shows the data hierarchy using attributes, characteristics, services, and profiles defined previously in this section.

5 PSoC™ 6 application notes

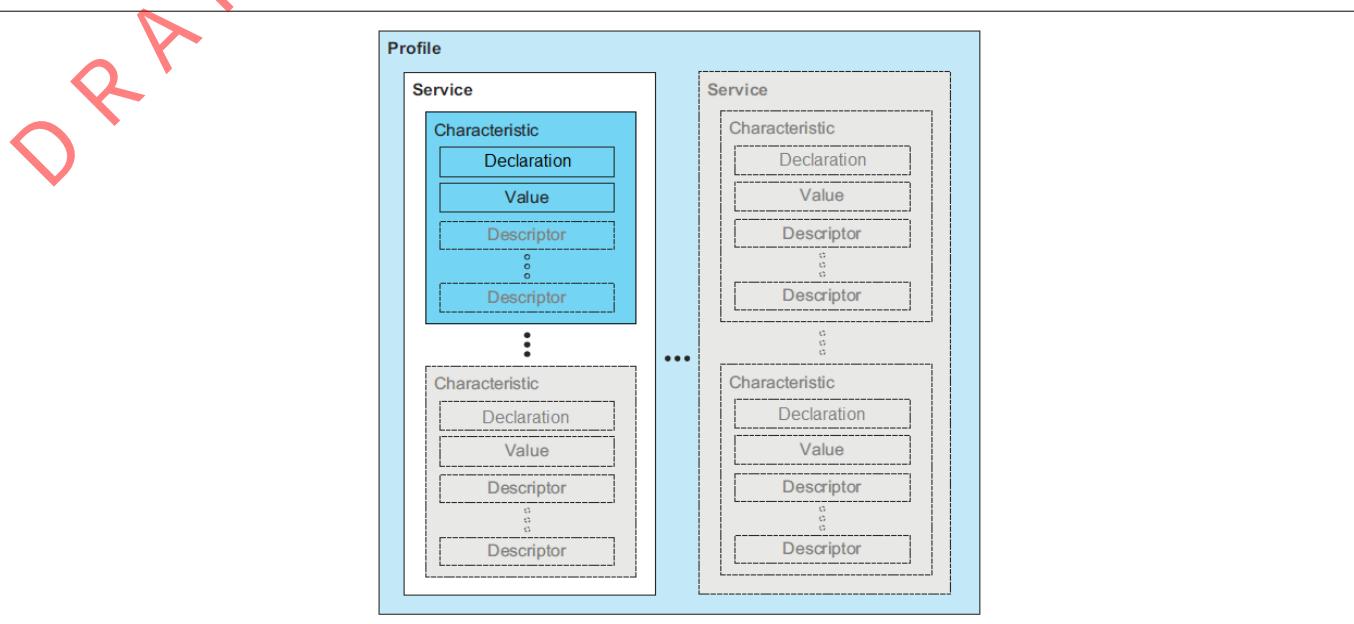


Figure 128 **Bluetooth® LE data hierarchy**

Note: *Image courtesy of Bluetooth® SIG.*

B.7.2 Attribute operations

Attributes defined in the previous section are accessed using the following five basic methods:

- **Read request:** The GATT Client sends this request to the GATT Server to read an attribute value. For every request, the GATT Server sends a response to the GATT Client. A smartphone reading the Battery-Level Characteristic of a heart-rate monitor device (see [Figure 126](#)) is an example of a Read Request
- **Write request:** The GATT Client sends this request to the GATT Server to write an attribute value. The GATT Server responds to the GATT Client, indicating whether the value was written. A smartphone writing a value of 0x01 to the CCCD of a Battery Level characteristic to enable notifications is an example of a Write Request
- **Write command:** The GATT Client sends this command to the GATT Server to write an attribute value. The GATT Server does not send any response to this command. For example, the Bluetooth® LE Immediate Alert Service ([IAS](#)) uses a Write Command to trigger an alert (turn on an LED, ring a buzzer, drive a vibration motor, and so on) on an IAS Target device (for example, a Bluetooth® LE key fob) from an IAS locator (for example, a smartphone)
- **Notification:** The GATT Server sends this to the GATT Client to notify it of a new value for an attribute. The GATT Client does not send any confirmation for a notification. For example, a heart-rate monitor device sends heart-rate measurement notifications to a smartphone when its CCCD is written with a value of 0x01
- **Indication:** The GATT Server sends this type of message. The GATT Client always confirms it. For example, a Bluetooth® LE Health Thermometer Service ([HTS](#)) uses indications to reliably send the measured temperature value to a health thermometer collector, such as a smartphone

B.8 Generic Attribute Profile (GATT)

The GATT defines the ways in which attributes can be found and used. The GATT operates in one of two roles:

- **GATT Client:** The device that requests the data (for example, a smartphone)
- **GATT Server:** The device that provides the data (for example, a heart-rate monitor)

5 PSoC™ 6 application notes

~~DRAFT~~

Figure 129 shows the client-server architecture in the GATT layer using a heart-rate monitoring device as an example. The heart-rate monitoring device exposes multiple services (HRS, BAS, and Device Information Service); each service consists of one or more characteristics with a characteristic value and descriptor, as shown in Figure 126.

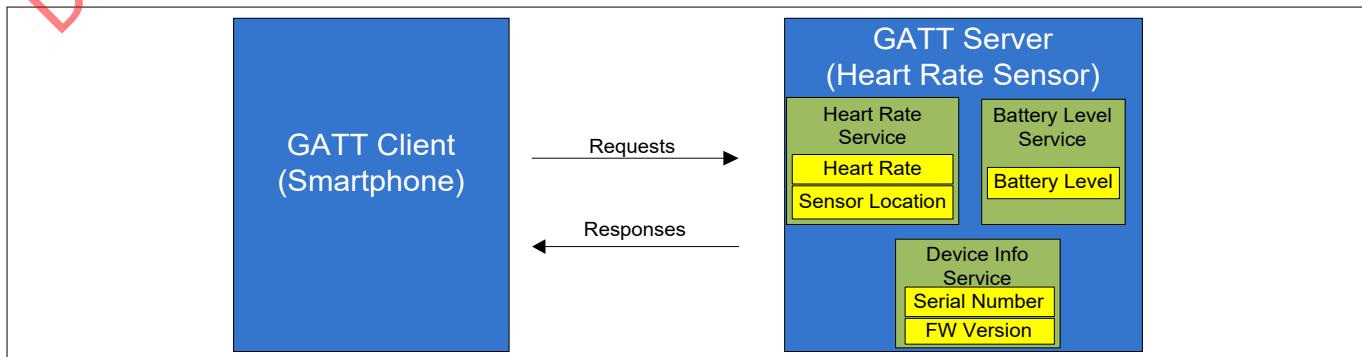


Figure 129 GATT Client-Server architecture

After the Bluetooth® LE connection is established at the link-layer level, the GATT Client (which initially knows nothing about the connected Bluetooth® LE device) initiates a process called “service discovery.” As part of the service discovery, the GATT Client sends multiple requests to the GATT Server to get a list of all the available services, characteristics, and attributes in the GATT Server. When service discovery is complete, the GATT Client has the required information to modify or read the information exposed by the GATT Server using the attribute operations described in the previous section.

B.9 Generic Access Profile (GAP)

The GAP layer provides device-specific information such as the device address; device name; and the methods of discovery, connection, and bonding. The Profile defines how a device can be discovered, connected, the list of services available, and how the services can be used. Figure 131 shows an example of a Heart Rate Profile.

The GAP layer operates in one of four roles:

- **Peripheral:** This is an advertising role that enables the device to connect with a GAP Central. After a connection is established with the central, the device operates as a slave. For example, a heart-rate sensor reporting the measured heart-rate to a remote device operates as a GAP peripheral
- **Central:** This is the GAP role that scans for advertisements and initiates connections with peripherals. This GAP role operates as the master after establishing connections with peripherals. For example, a smartphone retrieving heart-rate measurement data from a peripheral (heart-rate sensor) operates as a GAP central
- **Broadcaster:** This is an advertising role that is used to broadcast data. It cannot form Bluetooth® LE connections and engage in data exchange (no request/response operations). This role works similar to a radio station in that it sends data continuously whether or not anyone is listening; it is a one-way data communication. A typical example of a GAP broadcaster is a beacon, which continuously broadcasts information but does not expect any response
- **Observer:** This is a listening role that scans for advertisements but does not connect to the advertising device. It is the opposite of the broadcaster role. It works similar to a radio receiver that can continuously listen for information but cannot communicate with the information source. A typical example of a GAP Observer is a smartphone app that continuously listens for beacons

Figure 130 shows a generic Bluetooth® LE system with Bluetooth® LE pioneer kit as the peripheral and a smartphone as the central device. The interaction between Bluetooth® LE protocol layers and their roles on the Central and the Peripheral devices are also shown.

5 PSoC™ 6 application notes

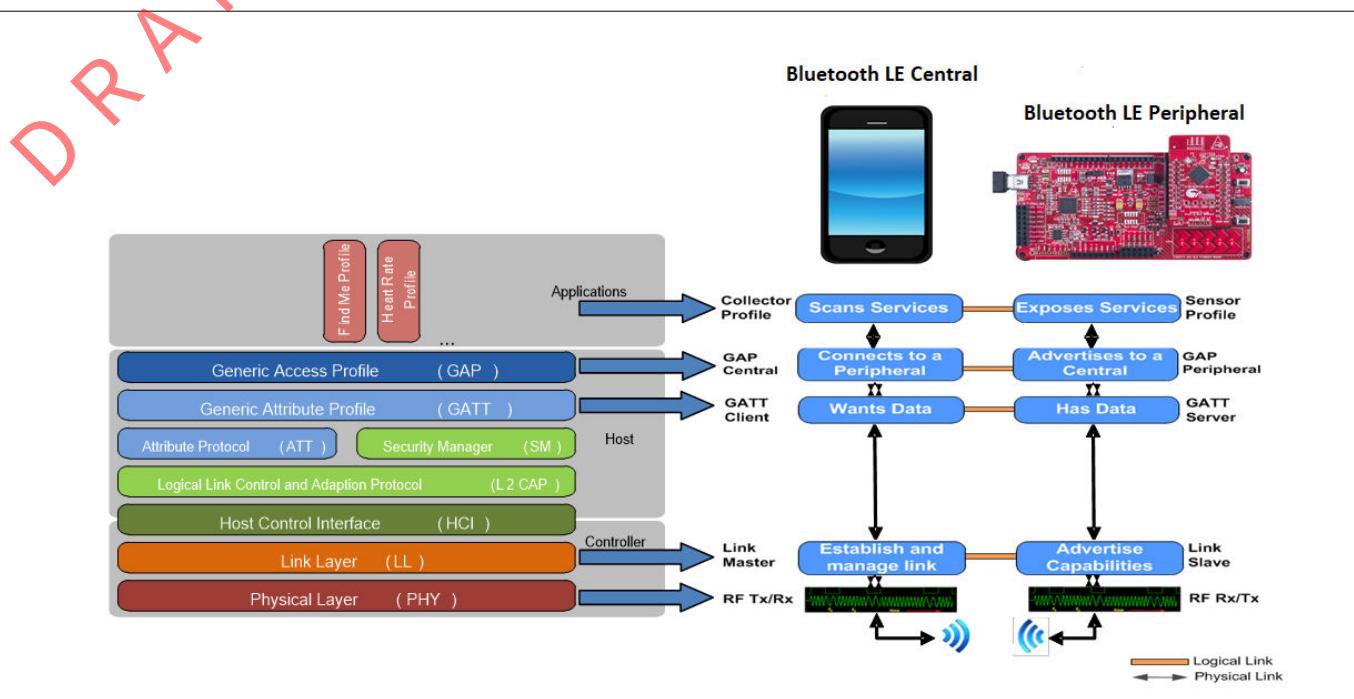


Figure 130 Bluetooth® LE system design

Figure 131 shows an example where a smartphone with a heart-rate app operates as a central and a heart-rate sensor operates as a peripheral. The heart-rate monitoring device implements the Heart-Rate Sensor Profile, while the smartphone receiving the data implements the Heart-Rate Collector profile.

In this example, the Heart-Rate Sensor Profile implements two standard services. The first is a Heart Rate Service that comprises three characteristics (the Heart Rate Measurement Characteristic, the Body Sensor Location Characteristic, and the Heart Rate Control Point Characteristic). The second service is a Device Information Service. At the link layer, the heart-rate measurement device is the slave and the smartphone is the master. See the Bluetooth® developer portal for a detailed description of the Heart Rate Service and Profile.

5 PSoC™ 6 application notes

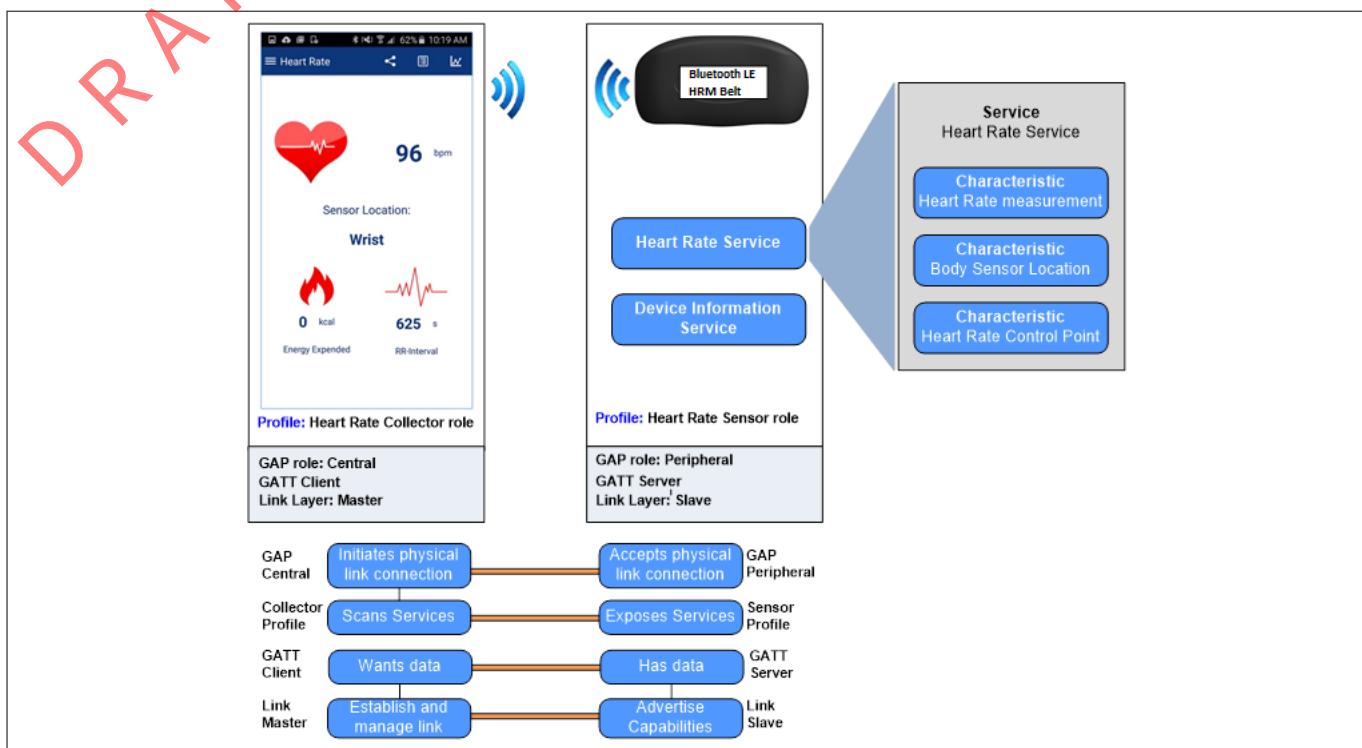


Figure 131 **Bluetooth® LE Heart-rate monitor system**

5 PSoC™ 6 application notes**C Appendix C. Device features****C.1 System wide resources****C.1.1 CPU subsystem: CM4 and CM0**

The CPU subsystem in PSoC™ 6-BLE consists of two Cortex® cores: CM4 with a single-precision floating-point unit capable of operating at a maximum frequency of 150 MHz; CM0+ capable of operating at a maximum frequency of 100 MHz. There is a memory protection unit (MPU) available in both the cores. Additionally, there are protection units attached to peripherals called Peripheral Protection Unit (PPUs) and Shared Memory Protection Units (SMPUs) for shared memory regions.

The CM0+ also provides a secure boot function. This allows system integrity to be checked and privileges enforced prior to the execution of the user application.

C.1.2 IPC

Inter-processor communication (IPC) provides the functionality for the two cores to communicate and synchronize their activities. IPC hardware is implemented using register structures in PSoC™ 6-BLE. These register structures are used to synchronize events, and trigger “notify” or “release” events of an IPC channel. PSoC™ 6-BLE supports up to 16 channels, which allows message passing between CPU cores and supports locks for mutual exclusion.

C.1.3 Memory system

The CPU cores have a fixed memory address map that enables shared access to memory and peripherals. Code can be executed from both the flash and RAM on both cores.

The PSoC™ 6-BLE family has up to 1 MB of flash memory and an additional 32 KB of flash that can be used for EEPROM emulation. There is also an additional 32 KB of supervisory flash. In addition, the flash supports Read-While-Write (RWW) operation so that the flash can be written to when the CPU is actively executing instructions. There is also a 128 KB ROM that contains boot and configuration routines. This will ensure Secure Boot operation if authentication of user flash is required for end applications.

PSoC™ 6-BLE has up to 288 KB of SRAM memory, which can be fully retained or retained in increments of user-designated 32 KB blocks.

C.1.4 DMA

The CPU subsystem includes two independent DMA controllers, each capable of running 32 channels. The controllers support independent accesses to the peripherals using the Arm® standard Advanced microcontroller Bus Architecture (AMBA) High-Performance Bus (AHB).

The DMA transfers data to and from memory, peripherals, and registers. These transfers occur independent of the CPU. The DMA channels support the following:

- 8-bit, 16-bit, and 32-bit data widths at both source and destination
- Four priority levels on each channel
- Configurable interrupts on each DMA descriptor
- Descriptor chaining

~~DRAFT~~ 5 PSoC™ 6 application notes

C.1.5 Clocking system

PSoC™ 6-BLE has the following clock sources:

- Internal main oscillator (IMO): The IMO is the primary source of internal clocking in PSoC™ 6-BLE. The CPU and all high-speed peripherals can operate from the IMO or an external crystal oscillator (ECO). PSoC™ 6-BLE has multiple peripheral clock dividers operating from either the IMO or the ECO, which generate clocks for high-speed peripherals. The IMO can generate an 8 MHz clock with an accuracy of ± 1 percent and is available only in Active mode.
- External crystal oscillator (ECO): The PSoC™ 6 MCU device contains an oscillator to drive an external 4 MHz to 33.33 MHz crystal for an accurate clock source

In addition, the external crystal oscillator on the Bluetooth® Low Energy subsystem with a built-in tunable crystal load capacitance is used to generate a highly accurate 32 MHz clock. It is primarily used to clock the Bluetooth® Low Energy subsystem that generates the RF clocks. The high-accuracy ECO clock can also be used as a clock source for the PSoC™ 6-BLE device's high-frequency clock (CLK_HF) and is designated as the AltHF clock.

- External clock (EXTCLK): The external clock is a megahertz-range clock that can be sourced from a signal on a designated I/O pin. This clock can be used as the source clock for either the PLL or FLL, or it can be used directly by the high-frequency clocks
- Internal low-speed oscillator (ILO): The ILO is a very-low-power 32 kHz oscillator, which primarily generates clocks for low-speed peripherals operating in all power modes
- Precision internal low-speed oscillator (PILO): PILO is an additional source that can provide a more accurate 32.768 kHz clock than ILO. PILO works in Deep Sleep and higher modes
- Watch crystal oscillator (WCO): The 32.768 kHz WCO is used as one of the sources for low frequency clock tree (CLK_LF) along with ILO/PILO. WCO is used to accurately maintain the time interval for Bluetooth® LE advertising and connection events. Similar to ILO, WCO is also available in all modes except the Hibernate and Stop modes

The PSoC™ Bluetooth® LE clock generation system has phase-locked loop (PLL) and frequency-locked loop (FLL) blocks that can be used to generate high-frequency clocks (CLK_HF). These high-frequency clocks in turn drive the CPU core clocks and the peripheral clock dividers.

PSoC™ 6-BLE has five high-frequency root clocks (CLK_HF [0-4]). Each CLK_HF has a destination on the device such as peripherals like serial memory interface.

PSoC™ 6-BLE has clock supervisors on high-frequency clock path 0 (CLK_HF [0]) and WCO to detect if the clock has been stopped and can trigger an interrupt or a system reset or both.

C.1.6 System interrupts

PSoC™ 6-BLE supports interrupts and exceptions on both the CPU cores: CM4 and CM0+. The cores provide their own vector table for handling interrupts/exceptions. PSoC™ 6-BLE can support up to 139 interrupts on CM4 and 32 interrupts on CM0+. Up to 33 interrupts can wake the device from Deep Sleep power mode

C.1.7 Power supply and monitoring

The PSoC™ 6-BLE device family supports an operating voltage of 1.71 V to 3.6 V. It integrates a single-input multiple-output (SIMO) buck converter to power the blocks within the device. The core operating voltage is user selectable between 0.9 V and 1.1 V. The device family supports multiple power supply rails – V_{DDD} , V_{DDA} , V_{DDIO} , and V_{BACKUP} . Additionally, there are power supply rails for Bluetooth® LE radio operation – V_{DDR} . The availability of various supply rails/pins for an application will depend on the device package selected.

The device includes a V_{BACKUP} supply pin to power a small set of peripherals such as RTC and WCO. This rail is independent of all other rails and can exist even when other rails are absent. Because the power supply to these blocks comes from a dedicated V_{BACKUP} pin, these blocks continue to operate even when the device

~~5 PSoC™ 6 application notes~~

power is disconnected or held in reset. The RTC present in the backup domain provides an option to wake up the device from any power modes.

The PSoC™ 6-BLE family supports power-on reset (POR), brownout detection (BOD), overvoltage protection (OVP), and low-voltage detection (LVD) circuits for power supply monitoring and to implement fail-safe recovery.

For more information on the power supplies in PSoC™ 6-BLE, see the [PSoC™ 6 MCU: PSoC™ 63 with BLE architecture technical reference manual](#).

C.1.8 Power modes

The power modes supported by PSoC™ 6-BLE, in the order of decreasing power consumption, are:

- Active mode: This is the primary mode of operation. In this mode, all peripherals are active and available
- Low-Power Active mode: This mode is akin to the Active mode with most peripherals operating with limited capability; CPU cores are available. The performance tradeoffs include reduced operating clock frequency, high-frequency clock sources limited in frequency, and lower core operating voltage
- Sleep mode: In this mode, the CPU cores are in Sleep mode, SRAM is in retention, and all peripherals are available. Any interrupt wakes up either of the CPUs and returns the system to Active mode. CM4 and C-M0+ both support their own CPU Sleep modes, and each CPU can be in Sleep independent of the state of the other CPU. The device is said to be in Sleep mode when both the cores are in CPU sleep
- Low-Power Sleep mode: Most peripherals operate with limited capability; CPU cores are not available
- Deep Sleep mode: In Deep Sleep mode, all high-speed clock sources are turned OFF. This in turn makes high-speed peripherals unusable in Deep Sleep. Peripherals that operate on low-frequency clocks only are available
- Hibernate mode: Device and GPIO states are frozen, and the device resets on wakeup

You can use a combination of Sleep, Deep Sleep, and Hibernate modes in a battery-operated Bluetooth® LE system to achieve best-in-class system power with longer battery life. [Table 12](#) shows the dependency between PSoC™ 6-BLE system power modes and BLESS power modes. A check mark in a cell of [Table 12](#) indicates that the BLESS can perform the specified task when the system is in a given power mode. For example, BLESS can be in Deep Sleep mode while the system is in Low-Power Active mode. The retention label indicates that the BLESS operating context is retained when the system switches to Deep Sleep mode.

Note: *The number of commons and segments supported by a PSoC™ 6-BLE device varies based on the device family and device package. See the respective device datasheet for details.*

Table 12 PSoC™ 6-Bluetooth® LE power modes

BLESS modes	PSoC™ 6-Bluetooth® LE system power modes					
	Active	Low-Power Active	Sleep	Low-Power Sleep	Deep Sleep	Hibernate
Transmit	✓	✓	✓	✓	retention	OFF
Receive	✓	✓	✓	✓	retention	OFF
Idle	✓	✓	✓	✓	retention	OFF
Deep Sleep	✓	✓	✓	✓	retention	OFF

~~DO NOT DRAFT~~ 5 PSoC™ 6 application notes

C.2 Secure Boot

Secure booting involves authenticating the application flash images using a key-based security protocol defined by a market/application-specific standard. The secure boot process is implemented in SROM and a separate supervisory flash in the PSoC™ 6-BLE device. The boot code computes a checksum of the boot code and compares it to a value in eFuse. If these values do not match, the boot process fails. The boot code also enforces debug access restrictions as specified in eFuse. Secure boot is an optional feature and needs to be enabled by the end-user.

C.3 Programmable digital peripherals

C.3.1 UDB

UDBs are programmable logic blocks that provide functionalities similar to CPLD and FPGA blocks, as [Figure 132](#) shows. UDBs allow you to create a variety of digital functions such as timer, counter, PWM, pseudo random sequence (PRS), CRC, shift register, SPI, UART, I²S, and custom combinational and sequential logic circuits.

Each UDB has two programmable logic devices (PLDs), each with 12 inputs and 8 product terms. PLDs can form registered or combinational sum-of-products logic. Additionally, an 8 bit single-cycle arithmetic logic unit (ALU), known as a “datapath,” is present in each UDB. The datapath helps with the efficient implementation of functions such as timer, counter, PWM, and CRC.

UDBs also provide a switched digital signal interconnect (DSI) fabric that allows signals from peripherals and ports to be routed to and through the UDBs for communication and control.

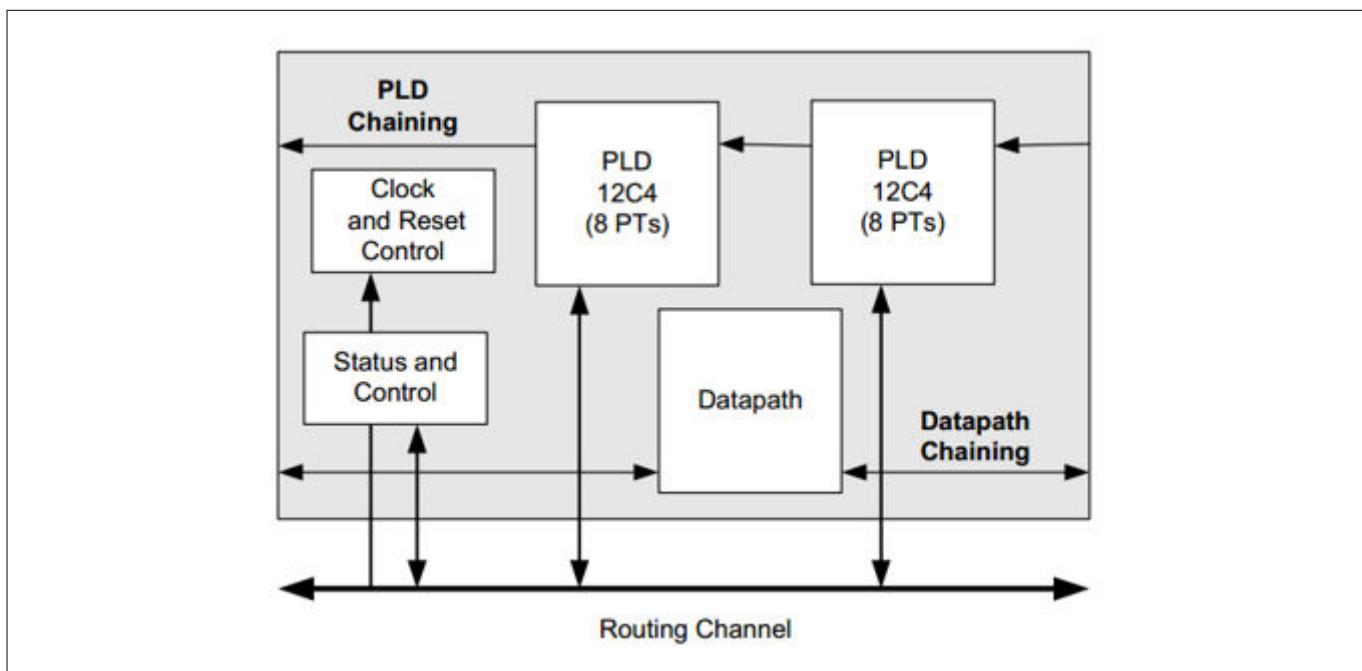


Figure 132 Universal digital block diagram

You do not necessarily need to know any hardware description language (HDL) to use UDBs. PSoC™ Creator, development tool for PSoC™ 6-BLE, can generate the required function for you from a schematic. If required, advanced users can implement custom logic on UDBs using Verilog.

PSoC™ 6-BLE has up to 12 UDBs. For more information on UDBs, see the following application notes:

- [AN62510 - Implementing state machines with PSoC™ 3, PSoC™ 4, and PSoC™ 5LP](#)

~~5 PSoC™ 6 application notes~~

- AN82156 - PSoC™ 3, PSoC™ 4, and PSoC™ 5LP - Designing PSoC™ Creator components with UDB datapaths
- AN82250 - PSoC™ 3, PSoC™ 4, and PSoC™ 5LP - Implementing programmable logic designs with verilog

~~C.3.2 Programmable TCPWM~~

PSoC™ 6-BLE has 32 programmable TCPWM blocks. Each TCPWM can implement a timer, counter, PWM, or quadrature decoder. TCPWMs provide dead band programmable complementary PWM outputs and selectable start, reload, stop, count, and capture event signals. The PWM mode supports center-aligned, edge, and pseudorandom operations. It also has a kill input to force outputs to a predetermined state.

For more information, refer to the PSoC™ 6-BLE TCPWM component datasheet.

C.3.3 SCB

PSoC™ 6-BLE has up to nine independent run-time programmable SCBs with I²C, SPI, or UART. The SCB supports the following features:

- Standard SPI master and slave functionality with Motorola®, Texas Instruments® Secure Simple Pairing (SSP), and National Semiconductor® Microwire protocols
- Standard UART functionality (up to 1 Mbps baud rate) with smart-card reader, single-wire local interconnect network (LIN) interface, SmartCard (ISO7816), and Infrared Data Association (IrDA) protocols
- Standard I²C master and slave functionality with operating speeds up to 1 Mbps
- EzSPI and EzI²C mode, which allows operation without CPU intervention
- One SCB can be configured to operate in Deep Sleep mode with an external clock. The low-power (Deep Sleep) mode of operation is supported on the SPI and I²C protocols (using an external clock) in slave mode only

For more information, see the PSoC™ 6-BLE SCB Component datasheet.

C.3.4 BLESS

PSoC™ 6-BLE incorporates a Bluetooth® smart subsystem that implements the BLE link layer and physical layer as specified in the Bluetooth® 4.2 specification. The Bluetooth® LE subsystem contains the physical layer (PHY) and link layer engine with an embedded AES-128 security engine. The subsystem supports all Bluetooth® SIG-adopted Bluetooth® LE profiles.

The physical layer consists of a digital PHY and RF transceiver compliant with the Bluetooth® 4.2 specification. The transceiver transmits and receives GFSK packets at 1 Mbps over the 2.4 GHz ISM band. The baseband controller is a composite hardware/firmware implementation that supports both master and slave modes. The key protocol elements such as HCI and link control are implemented in firmware, while time-critical functions such as encryption, CRC, data whitening, and access code correlation are implemented in hardware.

The BLESS is Bluetooth® 4.2 compliant with support for all the features of the Bluetooth® 4.0 specification and some additional features of the Bluetooth® 4.2 specification such as low-duty-cycle advertising, LE ping, L2CAP connection-oriented channels, link layer privacy, link layer data length extension, and LE secure connection. The BLESS block also contains an ECO and WCO that are required for generating an accurate RF frequency and keeping the time between successive connection intervals on the Bluetooth® LE link respectively.

The Bluetooth® LE subsystem supports up to four simultaneous connections. It supports its own four functional power modes: Deep Sleep, Idle, Transmit, and Receive.

Note: *The power modes discussed in this section are specific to the BLESS block. For PSoC™ 6-BLE system power modes, see the section [Appendix C.1.8](#).*

~~5 PSoC™ 6 application notes~~

~~Deep Sleep mode~~

Deep Sleep mode is the lowest power functional mode supported by the BLESS. In this mode, the radio is off. This mode is entered for maximum power saving during an advertising or connection interval after the packet transmission and reception is complete. The ECO can be turned off in this mode for power saving; the WCO, which is the low-frequency clock, is on for maintaining the BLE link layer timing reference logic. The application firmware controls the entry to and exit from this state.

Sleep mode

In Sleep mode, the radio is off. The block maintains all the configurations. The ECO and WCO are turned on, but the clock to the core BLESS logic is turned off. The application firmware controls the entry to and exit from this state.

Idle mode

The Idle mode is the preparation state for the transmit and receive states. In this state, the radio is turned off, but the link layer clock is enabled for the link layer logic so that the CPU starts the protocol state machines.

Transmit mode

Transmit mode is the active functional mode; all the blocks within BLESS are powered on. The link layer clock is enabled to complete the logic within the link layer and RF-PHY. In this mode, RF-PHY gets up to 2 Mbps of serial data from the link layer and transmits the 2.4 GHz GFSK-modulated data to the antenna port. BLESS enters Transmit mode from Idle mode.

Receive mode

This mode enables the BLESS to move into the receive state to perform BLE-specific receiver operations. RF-PHY translates the 1 Mbps data received from the RF analog block and forwards it to the link layer controller after demodulation. A summary of the BLESS power modes and operational sub-blocks is given in [Table 13](#).

Table 13 **BLESS power modes**

BLESS power mode	ECO	WCO	RF Tx	RF Rx	BLESS Core
Deep Sleep	Off	On	Off	Off	Off
Sleep	On	On	Off	Off	Off
Idle	On	On	Off	Off	On
Transmit	On	On	On	Off	On
Receive	On	On	Off	On	On

C.3.5 Audio subsystem

PSoC™ 6-BLE has an audio subsystem that consists of an I²S block and two PDM channels. The PDM channels interface to a digital microphone's bit-stream output. The PDM processing channel provides droop correction and can operate with clock speeds ranging from 384 kHz to 3.072 MHz and produce word lengths of 16 to 24 bits at audio sample rates of up to 48 ksps.

The I²S interface supports both master and slave modes with word clock rates of up to 192 ksps when 8-bit to 32-bit words are transmitted.

5 PSoC™ 6 application notes~~DRAFT~~
C.3.6 Serial Memory Interface

The Serial Memory Interface (SMIF) on PSoC™ 6-BLE is capable of interfacing with different types of memories and up to four memories. SMIF supports Octal-SPI (8 bits/cycle throughput), Dual quad-SPI (8 bits/cycle throughput), quad-SPI (4 bits/cycle throughput), Dual-SPI (2 bits/cycle throughput), and SPI (1 bit/cycle throughput). The block also supports execute-in-place (XIP) mode where the CPU cores can execute code directly from external memory. The SMIF block along with software modules developed in PSoC™ Creator enable you to use a qualified predetermined set of memory devices in your applications.

C.3.7 eFUSE

eFuse is a one-time programmable that is be used to program security-related settings. It can also be used to store your application settings that are programmed once and used later.

C.3.8 Segment LCD

PSoC™ 6-BLE has a segment LCD drive with the following features:

- Supports up to 8 common (COM) and 64 segment (SEG) electrodes
- Programmable GPIOs provide flexible selection of COM and SEG electrodes
- Supports 14-segment and 16-segment alphanumeric display, 7-segment numeric display, dot matrix, and special symbols
- Two drive modes: digital correlation and PWM
- Operates in all system power modes except Hibernate
- Can drive a 3 volt display from 1.8 volt VDD
- Digital contrast control

C.4 Programmable analog peripherals**C.4.1 Continuous Time Block Opamps**

PSoC™ 6-BLE devices have a pair of Continuous Time Block (CTBm) based opamps that have their inputs and outputs connected to fixed location pins. The opamps support all power modes except Hibernate. However, the opamps operate with reduced gain bandwidth product in Deep Sleep mode. The outputs of these opamps in typical usage can be used as buffers for the SAR inputs.

C.4.2 Low-Power comparator

PSoC™ 6-BLE devices have a pair of low-power comparators that can also operate in Deep Sleep and Hibernate modes. The comparators consume less than 300 nA of current in low-power modes. In a power-sensitive design, when the device goes into low-power mode, you can use the low-power comparator to monitor analog inputs and generate an interrupt that can wake up the system.

For more information, refer to the [PSoC™ 6-BLE Low-Power comparator component datasheet](#).

C.4.3 SAR ADC

PSoC™ 6-BLE has a 12-bit, 1 Msps successive approximation register (SAR) ADC with input channels that support programmable resolution and single-ended or differential input options. The number of GPIOs limits the

5 PSoC™ 6 application notes

number of ADC input channels that can be implemented. The SAR ADC does not operate in Deep Sleep and Hibernate modes as it requires a high-speed clock.

The SAR ADC has a hardware sequencer that can perform an automatic scan on as many as eight channels without CPU intervention. The SAR ADC can be connected to a fixed set of pins through an 8-input sequencer. The sequencer cycles through the selected channels autonomously (sequencer scan) and does so with zero switching overhead. It also supports preprocessing operations such as accumulation and averaging of the output data on these eight channels.

You can trigger a scan with a variety of methods, such as firmware, timer, pin, or UDB, giving you additional design flexibility.

To improve the performance in noisy conditions, it is possible to provide an external bypass on a fixed location pin for the internal reference amplifier. For more information on the SAR ADC, see the PSoC™ 6-BLE SAR ADC Component datasheet.

PSoC™ 6-BLE has an on-chip temperature sensor that can be used to measure temperature. The temperature sensor can be connected to the SAR ADC, which digitizes the reading and produces a temperature value by using software Component that includes calibration and linearization.

C.4.4 DAC

PSoC™ 6-BLE has a 12-bit voltage mode continuous time DAC (CTDAC), which has a settling time of 2 µs. The 12-bit DAC provides continuous time output without the need for an external sample and hold (S/H) circuit. The DAC control interface provides an option to control the DAC output through the CPU and DMA. This includes a double-buffered DAC voltage control register, clock input for programmable update rate, interrupt on DAC buffer empty to CPU, and trigger to DMA. The DAC may hence be driven by the DMA controllers to generate user-defined waveforms.

For more information on the DAC, see the PSoC™ 6-BLE DAC Component datasheet.

C.4.5 CAPSENSE™

The fourth-generation CAPSENSE™ in the PSoC™ 6-BLE device supports self-capacitance and mutual capacitance-based touch sensing. CAPSENSE™ is supported on all pins.

Capacitive touch sensors use human-body capacitance to detect the presence of a finger on or near a sensor. Capacitive sensors are aesthetically superior, easy to use, and have long lifetimes. The CAPSENSE™ feature in PSoC™ 6-BLE offers best-in-class SNR; best-in-class liquid tolerance; and a wide variety of sensor types such as buttons, sliders, track pads, and proximity sensors.

A software component makes capacitive sensing design very easy; the component supports an automatic hardware-tuning feature called SmartSense and provides a gesture recognition library for trackpads and proximity sensors.

For more information, see the [AN85951 - PSoC™ 4 and PSoC™ 6 MCU CAPSENSE™ design guide](#).

The CAPSENSE™ block has two 7-bit IDACs, which can be used for general purposes if CAPSENSE™ is not using them. A (slow) 10-bit slope ADC may be realized by using one of the IDACs. Refer to the PSoC™ 6-BLE CAPSENSE™ component datasheet for more information on how to use the slope ADC.

C.5 Programmable GPIOs

The I/O system provides an interface between the CPU cores, the peripherals, and the external devices. PSoC™ 6-BLE has up to 104 programmable GPIO pins. You can configure the GPIOs for CAPSENSE™, LCD, analog, or digital signals. PSoC™ 6-BLE GPIOs support multiple drive modes, drive strengths, and slew rates.

PSoC™ 6-BLE offers an intelligent routing system that gives multiple choices for connecting an internal signal to a GPIO. This flexible routing simplifies circuit design and board layout.

5 PSoC™ 6 application notes

Additionally, PSoC™ 6-BLE includes up to two Smart I/O ports, which can be used to perform Boolean operations on signals going to and coming from the GPIO pin. Smart I/O ports are also operational in Deep-Sleep mode.

DRAFT

5 PSoC™ 6 application notes

D Appendix D. IoT development tools

D.1 CY8CKIT-062-BLE PSoC™ 6-BLE pioneer kit

The PSoC™ 6-BLE pioneer kit shown in [Figure 133](#) is a Bluetooth® LE development kit that supports the PSoC™ 6-BLE family of devices.

Following are the features of the PSoC™ 6-BLE pioneer kit baseboard:

- Can be powered by a coin-cell battery or through the Type-C USB interface. The Type-C USB interface also supports up to 12 V, 3 A power delivery (PD) consumer and provider profiles
- Enables development of battery-operated low-power Bluetooth® LE designs that work in conjunction with standard, Arduino Uno connector-compliant shields or the onboard PSoC™ 6-BLE device capabilities, such as the CAPSENSE™ user interface and serial memory interface
- Supports third-party programming, debugging, and tracing with the Cortex® Debug/ETM connector
- Includes an additional header that supports interfacing with Pmod™ daughter cards from third-party vendors such as Digilent
- Supports PDM-PCM microphone for voice-over-Bluetooth® LE functionality
- Includes QSPI NOR flash and F-RAM

The kit includes the following:

- A USB-Bluetooth® LE dongle that acts as a Bluetooth® LE link master and works with the CySmart Host Emulation Tool to provide a Bluetooth® LE host emulation platform on non-Bluetooth® LE Windows PCs
- An E-Ink display

The kit consists of a set of BLE example projects and documentation that help you get started on developing your own Bluetooth® LE applications. Visit the [CY8CKIT-062-BLE PSoC™ 6-BLE pioneer kit](#) webpage to get the latest updates on the kit and download the kit design, example projects, and documentation files.

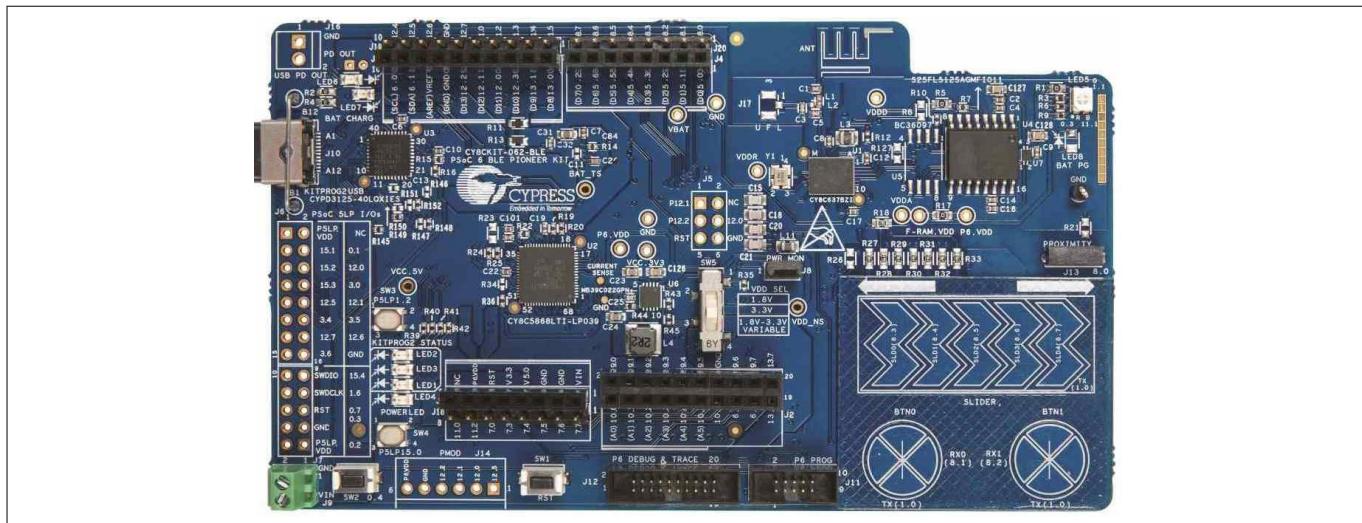


Figure 133 CY8CKIT-062-BLE PSoC™ 6-BLE pioneer kit

D.2 CySmart Host Emulation Tool

The CySmart Host Emulation Tool is a Windows application that emulates a Bluetooth® LE central device using the PSoC™ 6-BLE pioneer kit's dongle; see [Figure 134](#). It is installed as part of the Bluetooth® LE pioneer kit installation and can be launched by right-clicking the Bluetooth® Low Energy component. It provides a platform for you to test your PSoC™ 6-BLE peripheral implementation over GATT or L2CAP connection-oriented channels.

5 PSoC™ 6 application notes

~~DRAFT~~
by allowing you to discover and configure the Bluetooth® LE services, characteristics, and attributes on your Peripheral.

Operations that you can perform with the CySmart Host Emulation Tool include, but are not limited to:

- Scan Bluetooth® LE peripherals to discover available devices to which you can connect
- Discover available Bluetooth® LE attributes including services and characteristics on the connected Peripheral device
- Perform read and write operations on characteristic values and descriptors
- Receive characteristic notifications and indications from the connected Peripheral device
- Establish a bond with the connected Peripheral device using Bluetooth® LE Security Manager procedures
- Establish a Bluetooth® LE L2CAP connection-oriented session with the Peripheral device and exchange data per the Bluetooth® 4.2 specification
- Perform over-the-air (OTA) firmware upgrade of Bluetooth® LE peripheral devices

Figure 134 and Figure 135 show the user interface of the CySmart Host Emulation Tool. For more information on how to set up and use this tool, see the CySmart user guide from the **Help** menu.

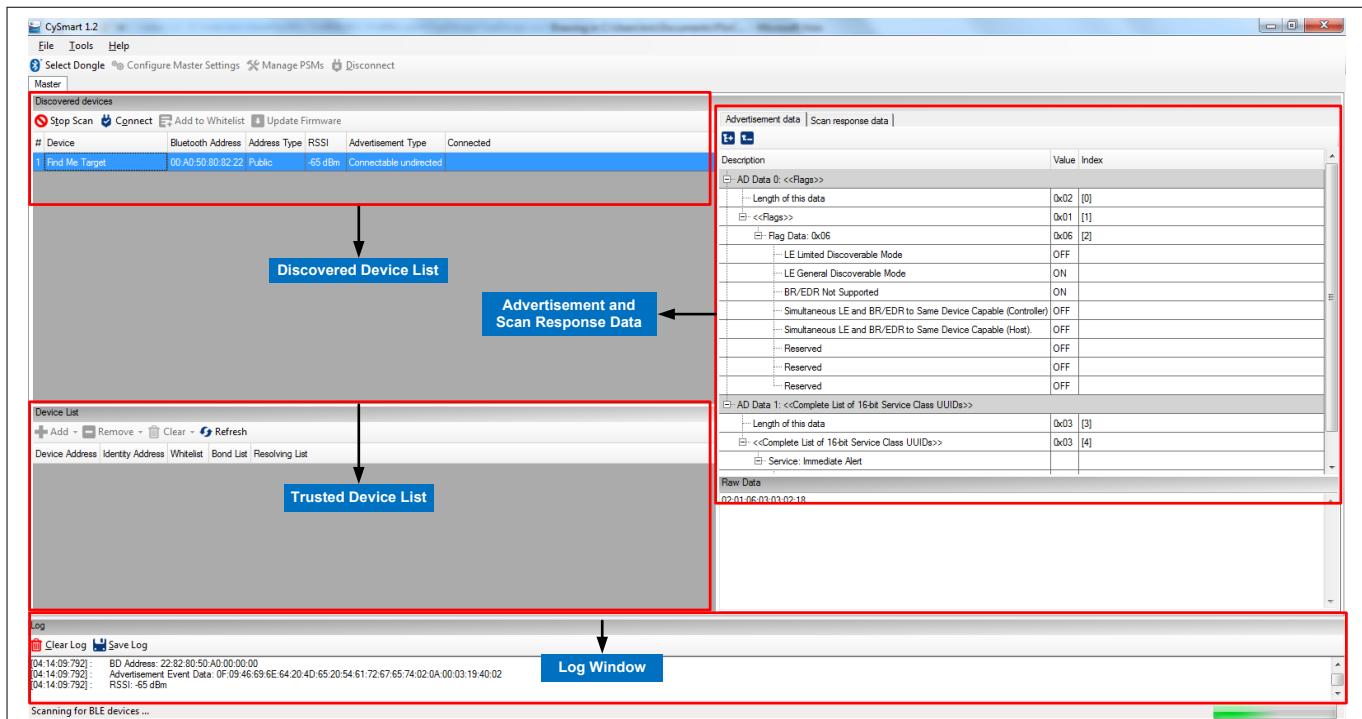


Figure 134 CySmart Host Emulation Tool Master device tab

5 PSoC™ 6 application notes

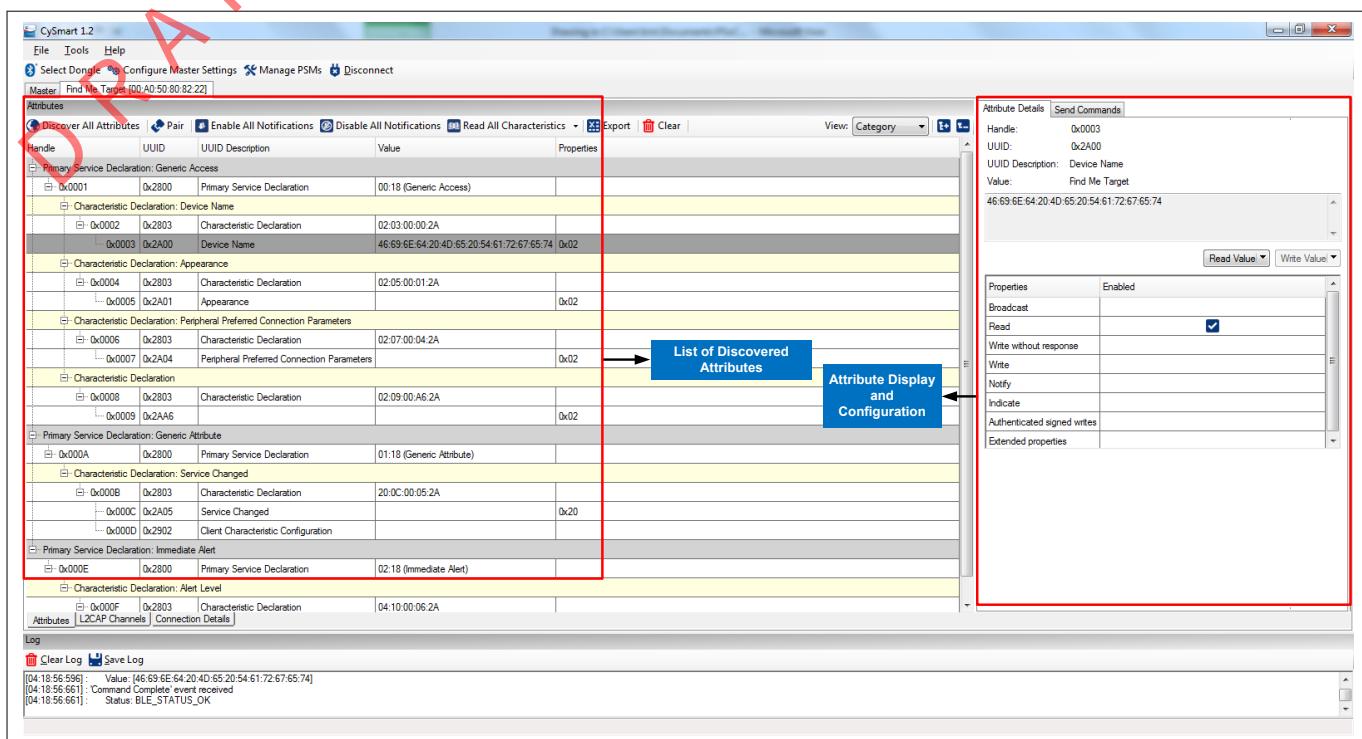


Figure 135 CySmart Host Emulation Tool peripheral device attributes tab

CySmart mobile app

In addition to the PC tool, you can download the CySmart mobile app for iOS or Android from the respective app stores. This app uses the iOS Core Bluetooth® framework and the Android built-in platform framework for Bluetooth® LE respectively. It configures your Bluetooth® LE -enabled smartphone as a Central device that can scan and connect to peripheral devices.

The mobile app supports SIG-adopted Bluetooth® LE standard Profiles through an intuitive GUI and abstracts the underlying Bluetooth® LE service and characteristic details. In addition to the Bluetooth® LE standard profiles, the app demonstrates a custom Profile implementation using LED and CAPSENSE™ demo examples. Figure 136 and Figure 137 show a set of CySmart app screenshots for the Heart Rate Profile user interface. For a description of how to use the app with Bluetooth® LE pioneer kit example projects, see the Bluetooth® LE pioneer kit guide.

5 PSoC™ 6 application notes

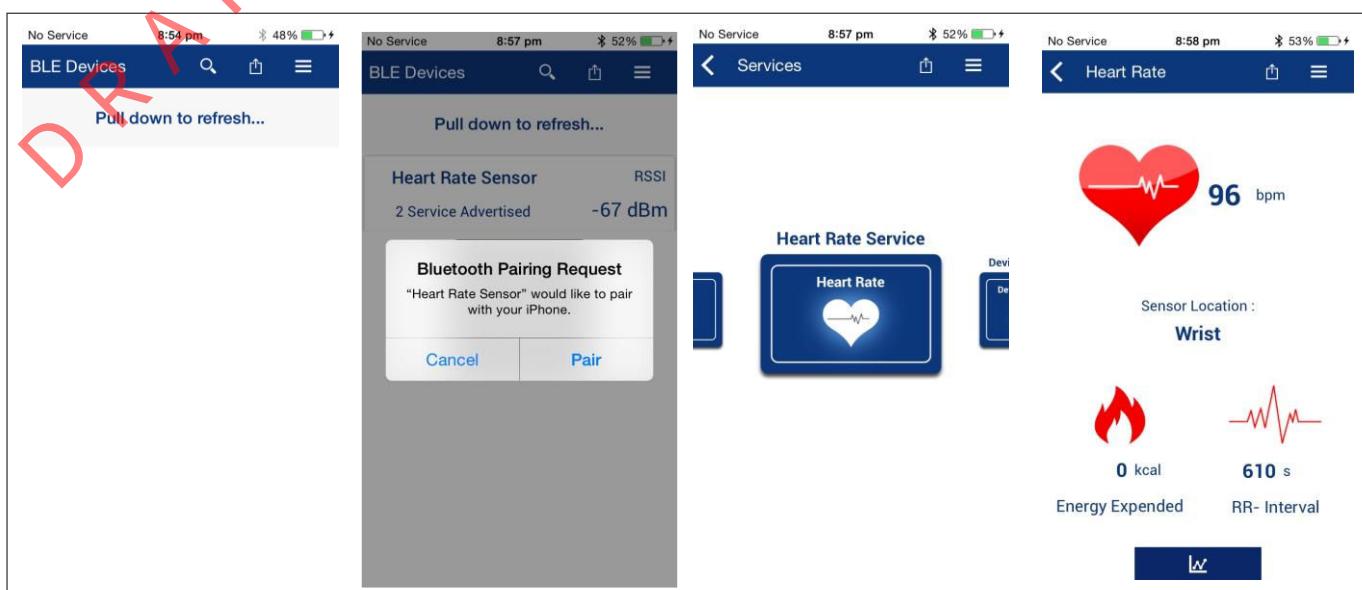


Figure 136 CySmart iOS App Heart Rate profile example

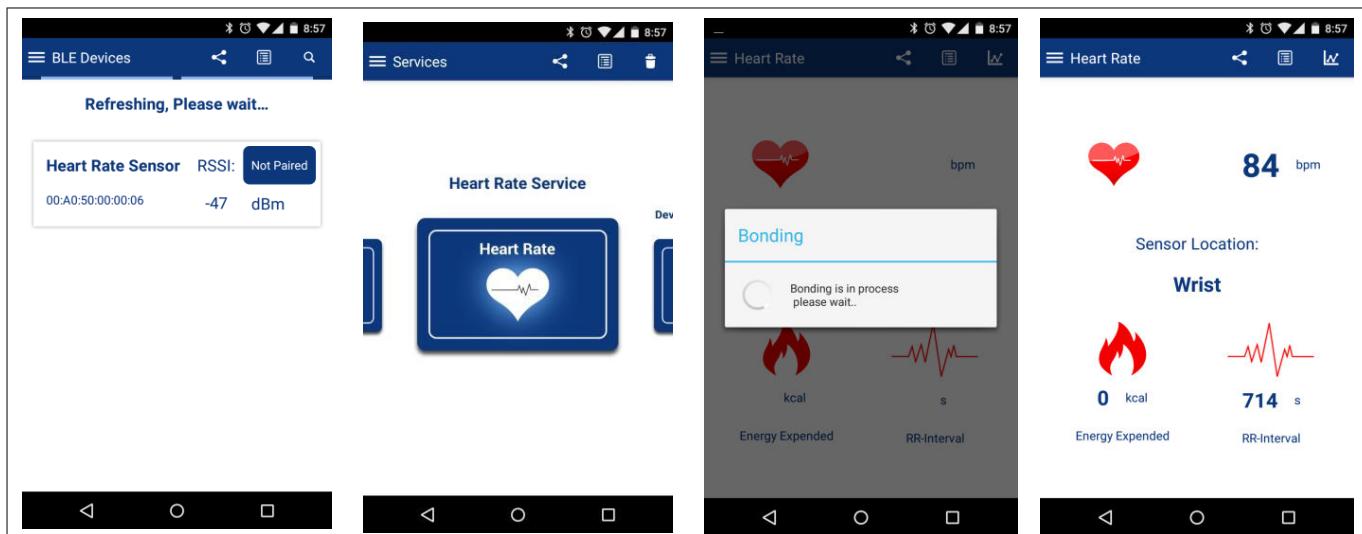


Figure 137 CySmart Android App Heart Rate profile example

5 PSoC™ 6 application notes**5.3.12 Revision history**

Document version	Date of release	Description of changes
**	2016-12-27	New Application Note
*A	2017-03-09	Rewrite of application note Updated UDB template
*B	2017-08-31	Updated title and figures
*C	2018-04-24	Partial rewrite of application note Updated template, abstract, and figures
*D	2019-10-11	Updated title
*E	2022-03-08	Migrated to Infineon template Replaced BLE with Bluetooth® LE/Bluetooth® Low Energy and PSoC™ 6 BLE to PSoC™ 6-BLE across the document Updated Software version as PSoC™ Creator 4.4 Fixed hyperlinks throughout the document

5.4 AN228753 PSoC™ 6 MCU usage of Direct Memory Access (DMA)**About this document**

-
- 4

Scope and purpose

This application note provides advanced guidance on DMA in PSoC™ 6 MCU and its use cases.

5 PSoC™ 6 application notes~~DO NOT USE~~

5.4.1 Introduction

A Direct Memory Access (DMA) block is specifically designed for data movement and is therefore more efficient than CPU for transferring large data blocks. In a system, DMA blocks also provide an independent data transfer engine, which relieves the CPU of bandwidth for data transfers. DMA blocks in PSoC™ 6 MCU implement a data transfer engine with different configurations and settings that let the DMA block be used in different data transfer use cases. This application note describes these different DMA configurations and use cases, and explains how to design with DMA to achieve the most efficient and fastest data transfers, and provides guidance for calculating the performance of a DMA transfer in the number of clock cycles.

DMA in PSoC™ 6 MCU is specifically designed to cater to data transfer requirements of a System on Chip (SoC). For this reason, there are two types of DMA engines in some of the PSoC™ 6 MCU devices.

DMA (DW) (also called Datawire): This DMA engine is specifically designed for transferring small data blocks, typically between peripherals, to offload the CPU from any data transfer with peripherals. In this document, this DMA engine will be referred to as DMA (DW) or DW.

DMAC (DMA Controller): This DMA engine is specifically designed to efficiently transfer large data blocks in memory. The DMAC provides higher performance compared to DMA/DW. In this document, this DMA engine is referred to as DMAC.

In addition to the two types of DMA hardware blocks, there are multiple other implementation-level features such as X and Y loops, different triggering schemes, and trigger routing. See the [Technical Reference Manual \(TRM\)](#).

This document will use the term DMA to generally refer to a feature or explanation for both DMA(DW) and DMAC.

~~5 PSoC™ 6 application notes~~

~~5.4.2~~ Architecture

There are two types of DMA hardware blocks implemented in PSoC™ 6 MCU devices: DMA/DW and DMAC. Both DMA/DW and DMAC have similar feature sets and architecture, but differ in their performance and use cases; see [Differences between DMA \(DW\) and DMAC](#). This section provides a general high-level description of the DMA block architecture. For a more detailed explanation of individual features, see the [Technical Reference Manual \(TRM\)](#).

The DMA transfer engine implements the state machine from the time a trigger is received, to when the transfer is completed. [Figure 138](#) shows the block diagram of the DMA hardware block.

The DMA hardware block can implement multiple DMA channels that can be independently configured for different data transfers. At a time, only one DMA channel can be active in a DMA block; other channels, if triggered, are put in a pending state. When the DMA hardware block completes the current active channel, pending channels are evaluated based on their priorities.

The data transfer associated with a DMA channel is described by a descriptor. The descriptor defines different configurations of the transfer such as size, data width, burst sizes, address increment schemes, and source and destination addresses. The descriptor is a structure of a specific type placed in a memory location. The pointer to this descriptor is associated with a DMA channel as part of its DMA channel configuration. When a DMA channel is made active, the first action is to fetch its descriptor from the memory. The descriptor could be in any form of memory: RAM, flash, or even an external memory. Multiple descriptors can be associated with a DMA channel in a chained configuration: see [Chaining descriptors](#).

The DMA hardware block connects to the system through two data bus interfaces and a set of trigger signals. The bus master interface is used for data transfers by the DMA block, while the bus slave interface is for other masters to access and configure DMA.

Each DMA channel has a trigger input, trigger output, and interrupt output line. Interrupt signals are routed to individual interrupt lines in the respective CPU. Trigger signals to and from DMA channels are routed through a trigger multiplexer block, which has a device-specific architecture. The trigger multiplexer block enables routing of trigger signals from different peripherals to the DMA block and routing trigger outputs back to other peripherals.

To understand specific trigger routes in a specific PSoC™ 6 MCU device, see the “Trigger Multiplexer Block” chapter in the [Technical Reference Manual \(TRM\)](#).

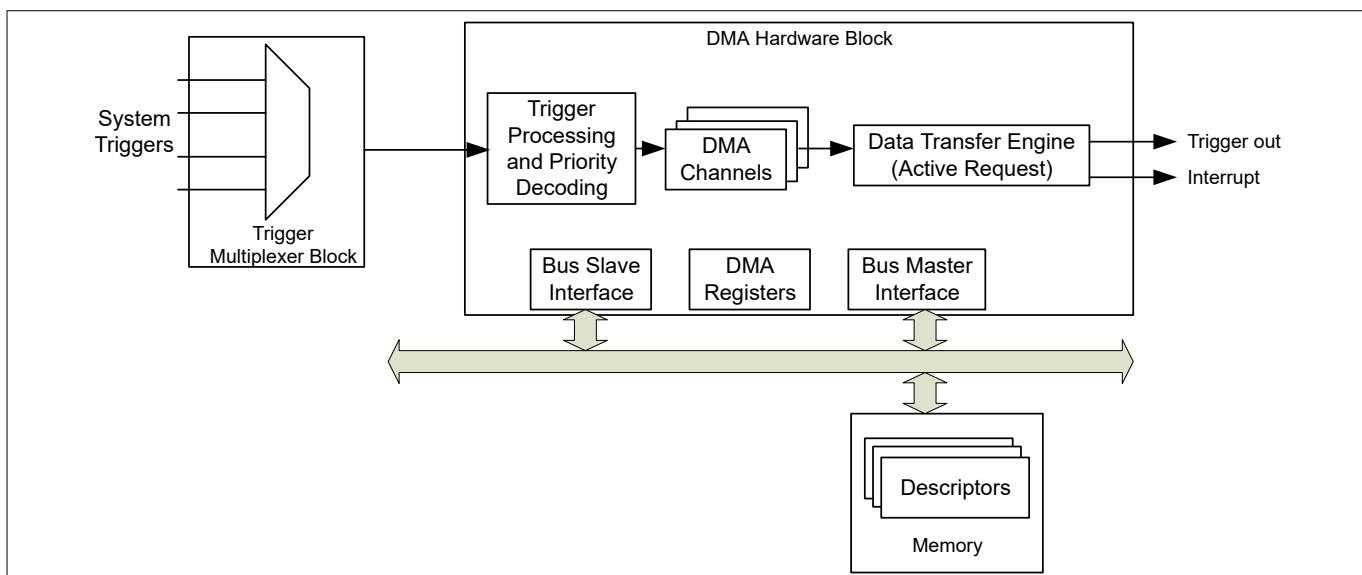


Figure 138 DMA architecture

~~DEAF~~

5 PSoC™ 6 application notes

5.4.2.1 Transfer modes

There are three transfer modes for a DMA channel as defined by its associated descriptor:

- In a **Single transfer**, the DMA descriptor transfers only a single data element. A data element could be a byte, 2 bytes, or a 4-byte word based on the width definition in the descriptor. Each single transfer would need to be initiated by a trigger signal to the DMA channel
- **1D transfer (X loop)** is useful for buffer-to-buffer transfer or a peripheral-to-memory buffer transfer; it allows for multiple data elements to be transferred as defined in a descriptor. The descriptor can decide on the exact form of source and destination address increments. You can choose to trigger the transfers one data element at a time or the entire 1D transfer at a time
- The **2D transfer (Y loop)** mode allows for multiple 1D transfers to be defined in a single descriptor. This allows for a larger data count and allows for transfers of more complex data entities like array of data structures. The trigger scheme allows for triggering individual data elements, an entire 1D transfer, or the entire 2D transfer at a time

~~5 PSoC™ 6 application notes~~

~~5.4.3 DMA design~~

Setting up a DMA channel using ModusToolbox™ involves multiple steps described in the following sections. The instructions assume that you have a basic understanding of bringing up a project using ModusToolbox™. See [AN228571 – Getting Started with PSoC™ 6 MCU on ModusToolbox™](#). The following sections take an example use case of an ADC triggering a DMA channel at the end of conversion. The DMA channel transfers the ADC result data to a memory buffer. The memory buffer is meant to be a 32-element-long array and therefore, the DMA descriptor will be configured to transfer the ADC result 32 times to fill the buffer array.

Steps for setting up a typical DMA channel are shown as numbered in [Figure 139](#).

5.4.3.1 Step 1: Choose the DMA channel

DMA channel trigger connections to peripherals are dependent on the design of the Trigger Multiplexer block. The choice of the DMA channel to use will depend on the trigger routing from the peripheral that is triggering the DMA channel. In this case, because the source of the DMA trigger is an ADC, you will need to refer the trigger routing and find the DMA channel associated with the ADC trigger outputs. In the case of the PSoC™ 62 CY8C62x8 or CY8C62xA device, DMA channel 28 provides this routing connection to the ADC as shown in [Figure 139](#). See the [Technical Reference Manual \(TRM\)](#) for a detailed discussion of Trigger Multiplexer block design.

After you identify the DMA channel, use the DMA tab in Device Configurator in ModusToolbox™, as shown in [Figure 139](#). Enable the DMA channel that you are planning to use, and then configure its parameters on the right of the screen. You can configure one or more DMA descriptors, DMA channel parameters, and the trigger input and output routing.

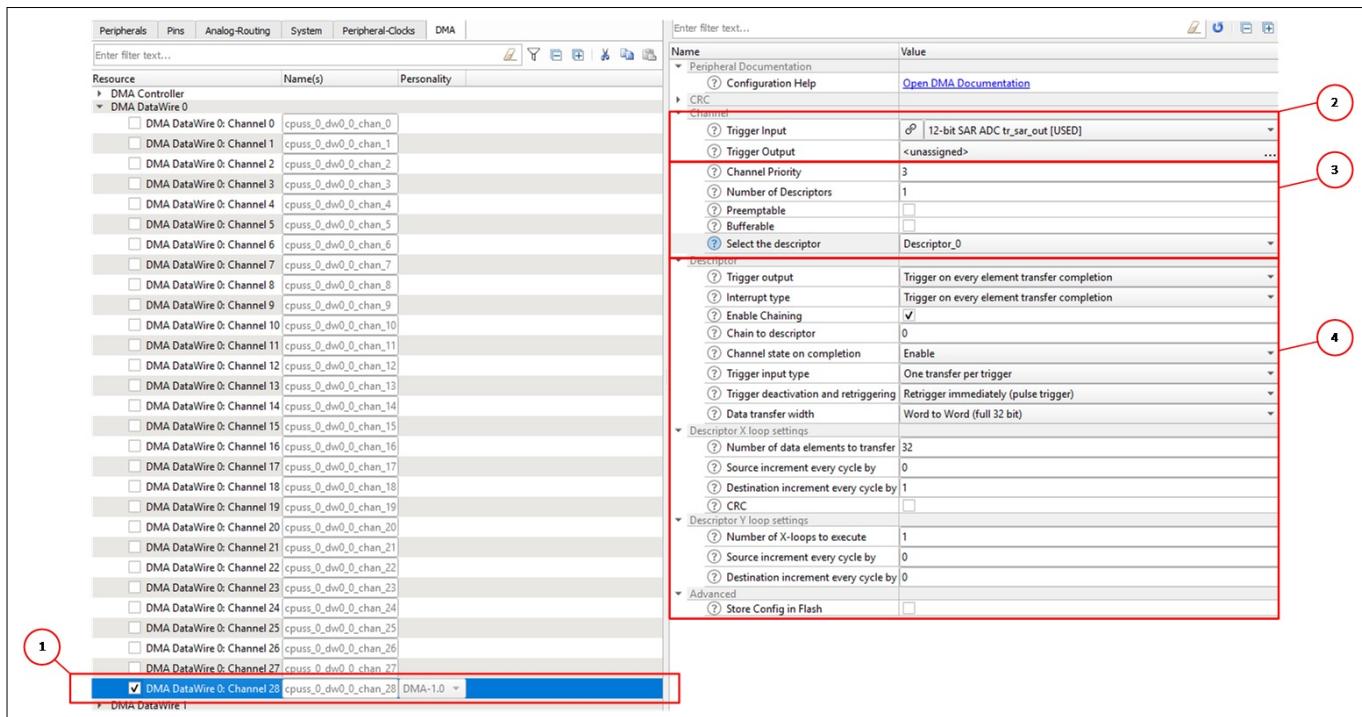


Figure 139 DMA configuration

5.4.3.2 Step 2: Configure triggers

The trigger input and outputs of a DMA channel are determined by the requirements of the system you are building. For the example of an ADC triggering the transfer of data from its result register to a memory buffer, the trigger input needs to be set to connect to the ADC trigger signal. You can configure the peripheral that will trigger the DMA channel by selecting the “Trigger Input” setting in the DMA channel settings. You can also

~~5 PSoC™ 6 application notes~~

configure how to route the trigger output from the DMA. Both these settings automatically configure the trigger multiplexer block routing.

5.4.3.3 Step 3: Set up the DMA channel

The channel configuration involves setting up channel-level parameters such as channel priority, preemptable, and bufferable. Here, you set the number of descriptors to be configured, and specify the descriptor that needs to be associated with the DMA channel when the channel is initialized.

5.4.3.4 Step 4: Set up the DMA descriptor

Based on the number of descriptors configured, the **Select Descriptor** field will have an option to select and configure as many descriptors as set in the **Number of Descriptors** field. Select each descriptor and configure it.

In the descriptor configuration, you can configure the trigger and interrupt behavior. In the case of multiple descriptors that need to be chained, select **Enable Chaining** and then select the descriptor to chain to in the **Chain to descriptor** field. You can also configure the X and Y loop transfers. Select **Store Config in Flash** to store the descriptors in flash.

5.4.3.5 Step 5: Write the user code

After configuring the triggers, channel, and descriptors, save and close the configurator. This automatically generates the code in the cycfg_DMAs.h and cycfg_DMAs.c files. Each descriptor will generate a descriptor structure called cpuss_0_dw0_0_chan_28_Descriptor_0 in the code. Note that this descriptor is not automatically initialized or allocated to the DMA channel. This must be done in user code.

1. A configuration structure named cpuss_0_dw0_0_chan_28_Descriptor_0_config is generated, which has all descriptor configuration set in the Device Configurator. This can be used to initialize the descriptor cpuss_0_dw0_0_chan_28_Descriptor_0
2. All channel-level configuration such as priority is configured in cpuss_0_dw0_0_chan_0_channelConfig. This structure can be used to initialize the channel

In addition, you need the following user code to initialize and enable DMA transfers:

1. Initialize the descriptor with the following function. This step transfers all configuration to the descriptor

```
Cy_DMA_Descriptor_Init(&cpuss_0_dw0_0_chan_0_Descriptor_0,
&cpuss_0_dw0_0_chan_0_Descriptor_0_config);
```

2. Configure the source and destination addresses with the following functions:

```
Cy_DMA_Descriptor_SetSrcAddress(cpuss_0_dw0_0_chan_0_Descriptor_0, SAR->CHAN_RESULT);
Cy_DMA_Descriptor_SetDstAddress(cpuss_0_dw0_0_chan_0_Descriptor_0, &Buffer);
```

3. Initialize the channel and associate the descriptor to it with the following function:

```
Cy_DMA_Channel_Init(cpuss_0_dw0_0_chan_0_HW, cpuss_0_dw0_0_chan_0_CHANNEL,
&cpuss_0_dw0_0_chan_0_channelConfig);
```

The initialization also automatically associates cpuss_0_dw0_0_chan_0_Descriptor_0 to the channel.

5 PSoC™ 6 application notes

4. Enable the channel using the following function:

```
Cy_DMA_Channel1_Enable(cpuss_0_dw0_0_chan_0_HW, cpuss_0_dw0_0_chan_0_CHANNEL)
```

5. Note that at this stage, only the channel is enabled but not the DMA block itself. To enable the DMA block, use the following function:

```
Cy_DMA_Enable(cpuss_0_dw0_0_chan_0_HW);
```

~~DRAFT~~

5 PSoC™ 6 application notes

5.4.4 Priorities and preemption

A single DMA hardware block can support many DMA channels which may be triggered by independent and unrelated events. This leads to the possibility of multiple DMA channels going to a pending state at the same time and contesting for bus access.

Every DMA channel has an associated priority value, which is used by the DMA hardware block's priority decoder to choose the channel when multiple channels are pending in the same DMA block. In such a situation, the DMA channel with the lowest priority number gets active and others are held pending. There are only four priority levels (0-3), while there can be more than four DMA channels present. In such cases, where multiple DMA channels of the same priority level are contesting for the bus, a round robin scheme of arbitration is employed.

However, if there is a low-priority channel already active and is in the middle of a large transfer, a pending higher-priority channel cannot become active. This can hold the execution of a higher-priority channel. This can be a problem when the higher-priority channel caters to a data transfer that is time-sensitive. [Figure 140](#) shows this condition in the case of Channel Without Preemption.

To address this, there is an additional configuration parameter in the DMA channel called "Preemptable". This parameter allows a higher-priority channel to preempt the currently active low-priority channel. If a channel has Preemptable enabled, any other channel with a higher priority can preempt the channel. This means that even when the DMA channel is in the middle of a transfer, a higher-priority channel request will stop the current transfer after completing the current atomic transfer, keep the channel pending, and then start the higher-priority channel. The Data Width parameter determines the size of an atomic transfer. Only when the high-priority channel is completed, the low-priority channel may resume. A low-priority preemptable channel can get preempted multiple times during a single transfer. See [Figure 140](#).

Preemption is useful only in cases when there are low-priority channels with no constraints of transfer times and there are high-priority channels that are time-sensitive and cannot be held hostage by other on-going transactions.

Note that preemption can cause significant delay in low-priority transfers, if:

- There are multiple frequent requests from high-priority channels
- The time taken for a high-priority transfer is too long

5 PSoC™ 6 application notes

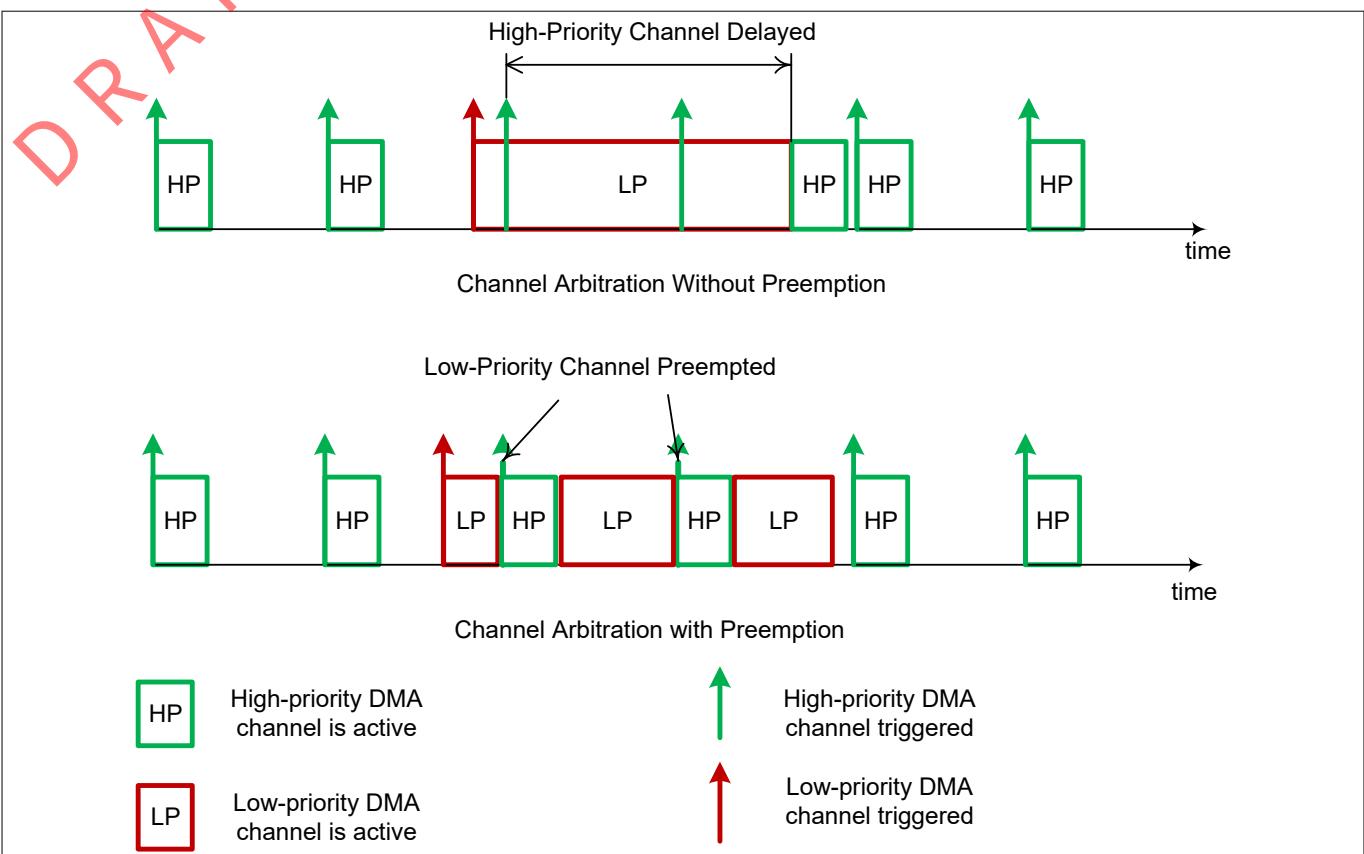


Figure 140 Channel arbitration and preemption in DMA

5 PSoC™ 6 application notes~~DO NOT USE~~
5.4.5 Data transfer widths

The data width determines the width of the data being accessed at the source or destination. This setting is also responsible for the value of each increment of the X or Y loops. Data widths must always be equal to the width supported by the device. For example, because all peripherals support 32-bit data width, if the source or destination of a transfer is a peripheral, the source or destination data width must be set to 32 bits.

Memory supports 8-bit, 16-bit, and 32-bit access. You can use larger data widths to increase throughputs, or use smaller data widths to quantize the data size. For example, 8-bit data is being transferred from a communication block to the memory, the source data width must be 32 bits (because the source is a peripheral), but the destination can be 8 bits (because the destination is a memory location), which automatically truncates the higher 24 bits. This will enable a smaller memory footprint.

5 PSoC™ 6 application notes

5.4.6 Types of transfers

5.4.6.1 1-to-1 transfer

This form of transfer allows for one data element to be transferred from a source to a destination. A good example is the peripheral-to-peripheral transfer shown in [Figure 141](#). This shows an SPI block and a UART block. The data coming on SPI Rx is directly transferred to UART Tx using DMA channel 0; similarly, the data coming on UART Rx is directly transferred to SPI Tx. The implementation utilizes the Rx interrupt of both the communication blocks to trigger DMA channels.

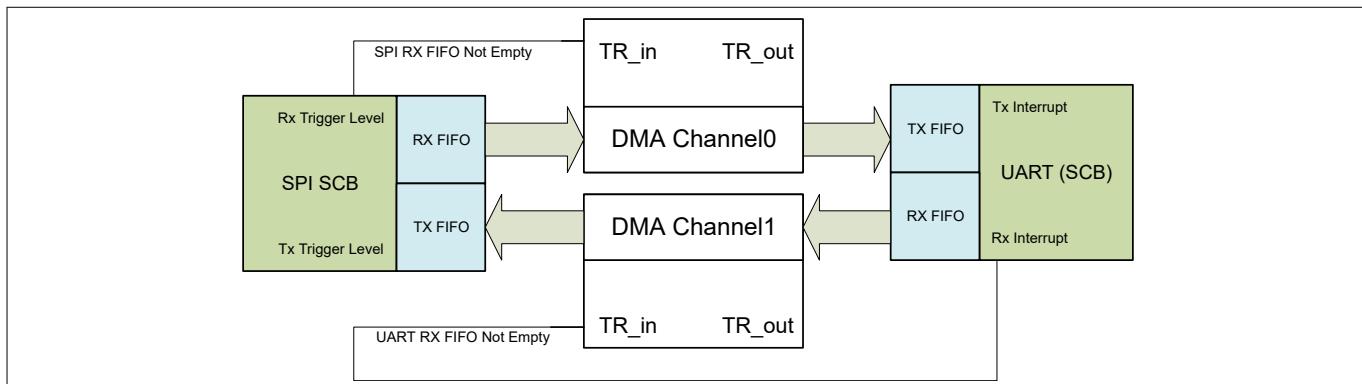


Figure 141 1-to-1 transfer

[Figure 142](#) shows the configuration for a 1-to-1 transfer. When the X and Y loops are configured to execute only once, the configurator automatically configures the DMA channel for a single transfer. This form of transfer is useful when a specific data transfer path is set up with a trigger, and is expected to work without CPU intervention.

The screenshot shows the PSoC Configurator interface for setting up a DMA descriptor. The descriptor is configured for a 1-to-1 transfer. The X loop settings are highlighted with a red box. The X loop settings include:

Number of data elements to transfer	1
Source increment every cycle by	0
Destination increment every cycle by	0
CRC	<input type="checkbox"/>

The Y loop settings are also shown:

Number of X-loops to execute	1
Source increment every cycle by	0
Destination increment every cycle by	0

Figure 142 Setting up a 1-to-1 transfer

~~5 PSoC™ 6 application notes~~

~~5.4.6.2~~ 1-to-N transfer

Transfer from a single source address to multiple destination addresses is a typical use case for transfers from peripherals like ADC to memory buffers for further processing by the CPU. The example in [Figure 143](#) shows an ADC output being DMA-transferred to a memory buffer. The ADC has a 12-bit resolution; therefore, the result is in a 16-bit data register, and the buffer is a 16-bit array. At the end of the entire buffer transfer, the DMA creates an interrupt for the CPU, so it can start post-processing on the buffered data.

The configuration for this example is shown in [Figure 144](#).

1. The trigger source is set to the SAR ADC, so the ADC trigger is automatically routed to the DMA channel
2. The interrupt is configured to trigger when the transfer is completed. This will enable the CPU to act on the buffer once the entire buffer is filled
3. The trigger input is configured to have one transfer in each ADC trigger. The data transfer width is configured to “Word to Halfword”. The ADC is a peripheral and therefore, the peripheral interface is 32-bit. The destination is a 16-bit buffer in memory
4. The X loop is set to run 10 times (which is the size of the buffer), while only the destination is incremented for each element transferred. The source remains constant because it is always the ADC result register

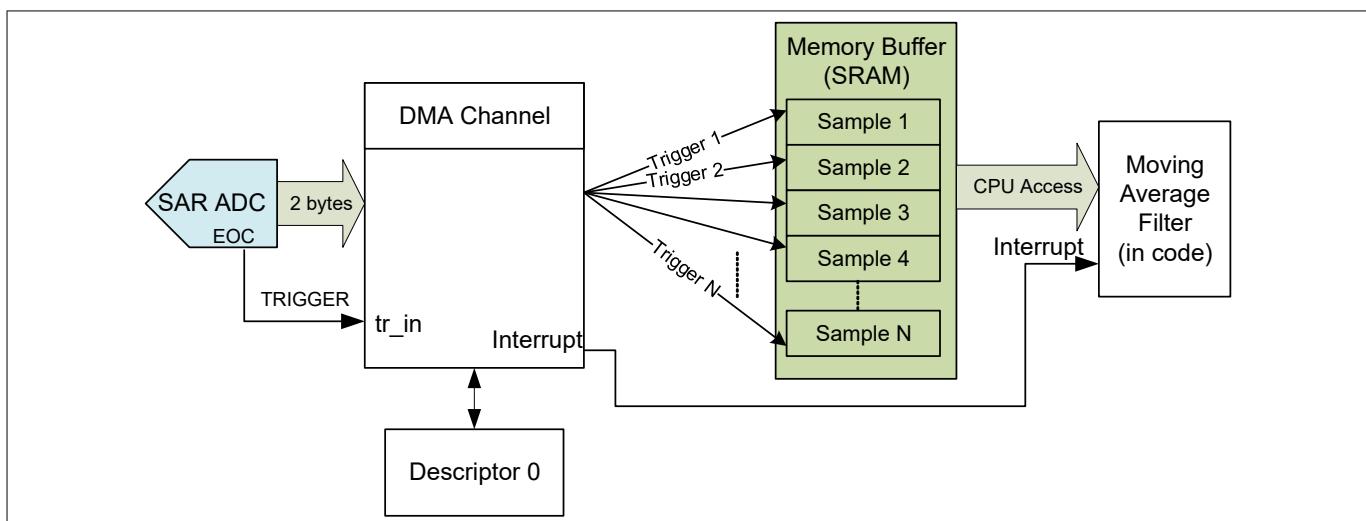


Figure 143 ADC to buffer: 1-to-N transfer configuration for ADC to buffer

5 PSoC™ 6 application notes

DRAFT

Trigger Input	12-bit SAR ADC tr_sar_out [USED]	1
Trigger Output	<unassigned>	...
Channel Priority	3	
Number of Descriptors	1	
Preemptable	<input type="checkbox"/>	
Bufferable	<input type="checkbox"/>	
Select the descriptor	Descriptor_0	
Descriptor		
Trigger output	Trigger on every element transfer completion	2
Interrupt type	Trigger on descriptor completion	
Enable Chaining	<input type="checkbox"/>	
Channel state on completion	Enable	3
Trigger input type	One transfer per trigger	
Trigger deactivation and retriggering	Retrigger immediately (pulse trigger)	
Data transfer width	Word to Halfword	
Descriptor X loop settings		
Number of data elements to transfer	10	4
Source increment every cycle by	0	
Destination increment every cycle by	1	

Figure 144 Configuration for ADC to buffer

5.4.6.2.1 Noncontiguous source/destination increments

In this transfer mode, you can also have the output placed in noncontiguous locations of the memory. This is useful if the destination was an array of structures and you are only filling a specific element in that array. An example is the destination that has an array of structures with each structure element comprising a 32-bit time stamp followed by a corresponding ADC result reading. This implementation is shown in [Figure 145](#). If a DMA channel was transferring only time stamp data, it would need to interleave destination addresses. This is accomplished by having a destination increment value greater than 1. The following use case independently transfers the time stamp data from its timer source, and the ADC result from the ADC.

5 PSoC™ 6 application notes

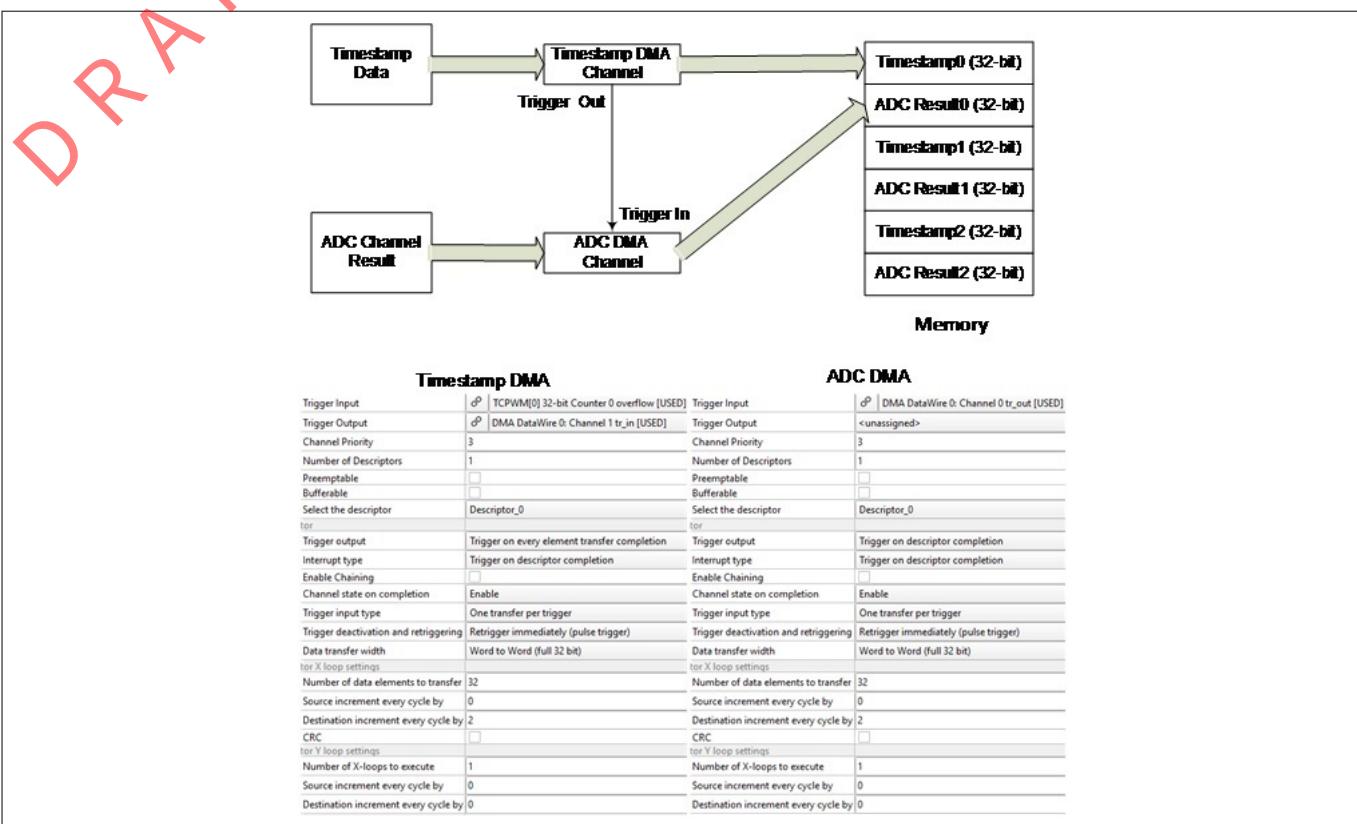


Figure 145 Example of transferring to noncontiguous destination addresses

The transfer can be accomplished by using two DMA channels with the time stamp channel triggering the ADC channel. In this case, the X loop settings will use a destination increment of 2. This will accomplish placing every result in the right interleaved location in the array. The same setting is used for both DMA channels.

5.4.6.3 N-to-1 transfer

This uses only an X loop transfer, but the source increments while the destination is constant, for example, when there is a waveform data stream in memory and it needs to be streamed to a DAC over SPI. Such an implementation will simply have non-zero increment for the source and a zero-increment value for the destination. The source can also be incremented by numbers greater than 1 for cases like the ones discussed in [1-to-N transfer](#).

5.4.6.4 N-to-N transfer

This is a use case for X loop when both source and destination addresses are incrementing such as a memory-to-memory transfer between flash and RAM for copying the configuration data to register blocks, or copying of a block of memory from the external to the internal memory.

5 PSoC™ 6 application notes

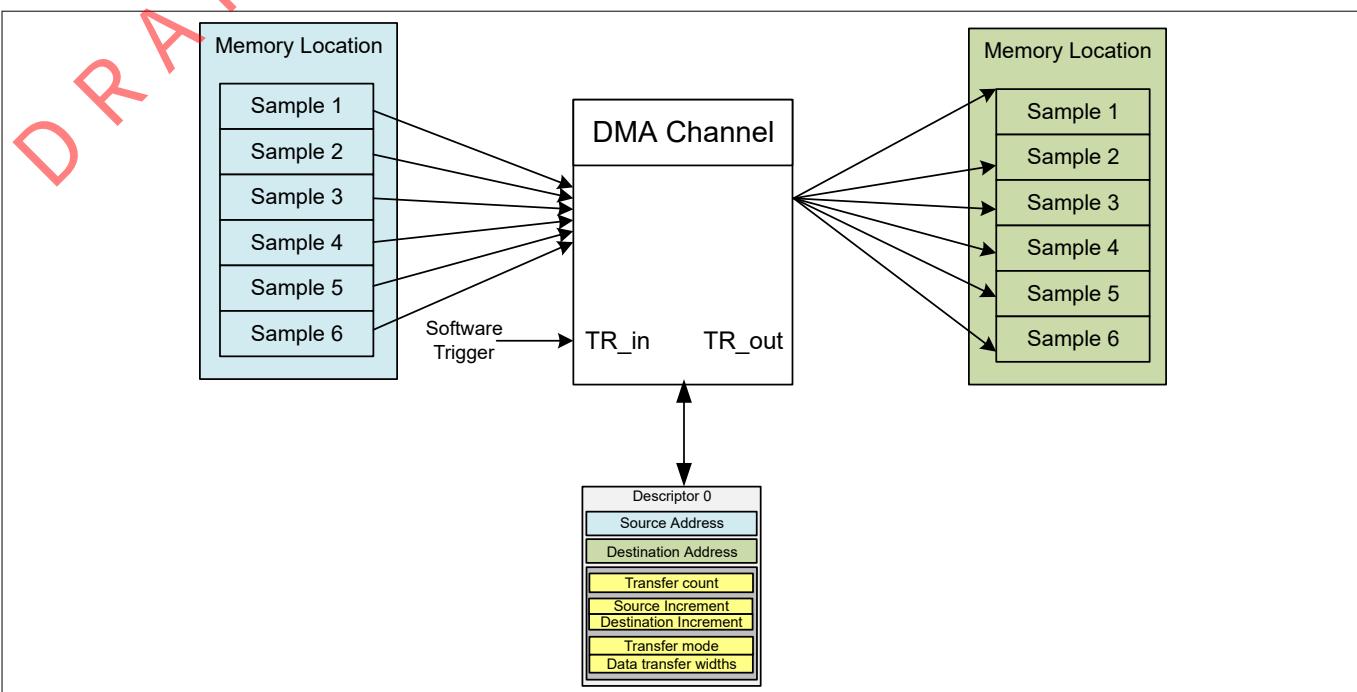


Figure 146 N-to-N transferN-to-N with dissimilar increment

Typically, such large data transfers between memories are time-critical. In such cases, you should use DMAC for better transfer speed.

Variations in this model are possible by using dissimilar source and destination increment values, such as shown in [Figure 147](#). Here, there is already mixed-in left and right channel audio data in the source memory.. They need to be separated into different buffers for audio processing. In this case, you can use two descriptors in a single channel. The first descriptor will transfer the left channel to its destination register while the second descriptor will transfer the right channel. However, each DMA descriptor will have source increments of 2 and destination increments of 1.

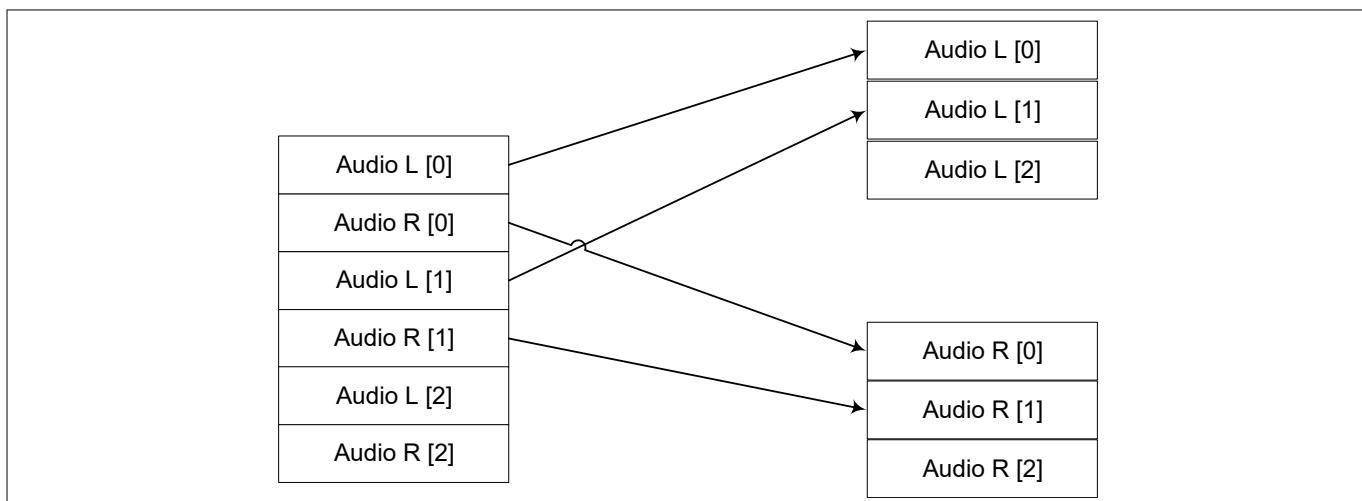


Figure 147 N-to-N with dissimilar increment

5.4.6.5 N-to-NxM

N-to-NxM is a case for using X and Y loops such as a structure that is getting transferred to an array of structures. Assume that there is a data structure coming over as a USB packet in every frame. This 60-byte long structure needs to be buffered into an array in the memory for further processing later as shown in [Figure 148](#).

5 PSoC™ 6 application notes

The configuration of X and Y loops for this transfer is shown in [Figure 149](#). Note that the source and destination are incremented in the X loop. This is the phase when the structure is being moved. Each X loop is used to move individual instances of the structure from USB to the buffer. After each X loop, the source address is reset back to the start of the source USB address because the Y source increment is zero. However, the Y loop destination increment is 60 (the size of the structure); this makes the next instance of the structure to get written as the next array element in the destination.

The scenario shown in [Figure 147](#) can also be implemented by using a N-to-NxM transfer with two X loops. The first time the X loop runs, it will transfer Audio L data from the source to the destination; the second time, it will transfer the Audio R data. The X loop increments will be two bytes on the source and one byte on the destination. However, the Y loop source increment will be 1. This is to ensure that after the first X loop, the next X loop starts at a '1' offset to get the R channel. The Y destination increment will be the distance between the Audio L and Audio R addresses.

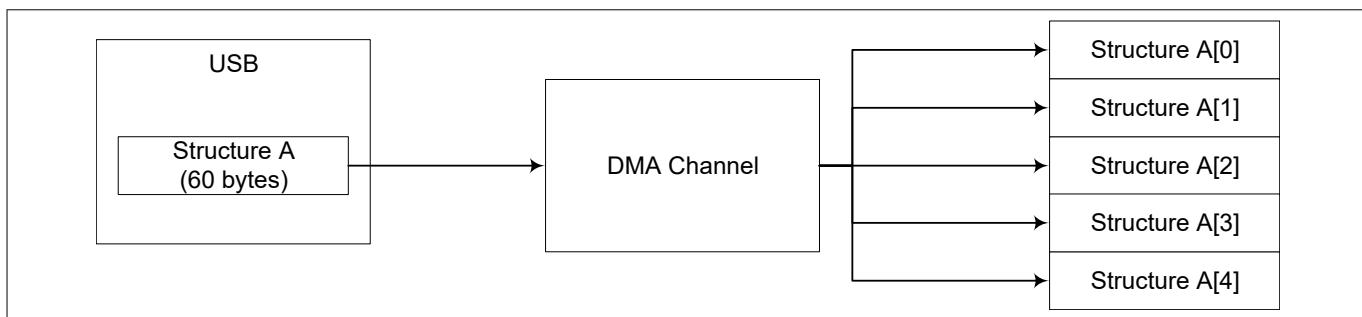


Figure 148 N-to-NxM transfer Configuration for an N-to-NxM transfer

Trigger Input	Universal Serial Bus (USB) 0 dma_req[0] [USED]
Trigger input type	One X loop transfer per trigger
Descriptor X loop settings	
Number of data elements to transfer	60
Source increment every cycle by	1
Destination increment every cycle by	1
CRC	<input type="checkbox"/>
Descriptor Y loop settings	
Number of X-loops to execute	5
Source increment every cycle by	0
Destination increment every cycle by	60

Figure 149 Configuration for an N-to-NxM transfer

5 PSoC™ 6 application notes~~DRAFT~~
5.4.7 Chaining descriptors

DMA blocks support descriptor chaining, which is useful if different types of transfers are to be done in a sequence. Each descriptor has the pointer to the next descriptor it must chain to, similar to a linked list. There is no limit on the number of descriptors you can chain. One of the greatest advantages of chaining is that each descriptor can have a different configuration including different source/destination addresses, trigger settings, interrupt settings, transfer modes, loops settings, and data widths. This allows the same DMA channel to implement multiple transfers of varying characteristics.

A good use case for descriptor chaining is double buffering as shown in [Figure 150](#). The input data comes in the form of 8-byte blocks in the SCB FIFO, which needs to be moved to Buffer 0 or 1 for double buffering. The buffers are 256 bytes each and therefore can accommodate 32 FIFOs worth of data before overflowing. A single DMA channel is used for the transfer, with two descriptors. Both descriptors are set up for a 2D transfer with the FIFO as the source. The X loop will cycle through the FIFO and therefore needs both source and destination increments. The Y loop handles the moving of 32 FIFOs. Descriptor 0 is set to chain to Descriptor 1 and vice versa. Descriptor 0 is configured to transfer to Buffer 0 and Descriptor 1 to Buffer 1.

Once Descriptor 0 is completed, the control automatically transfers to Descriptor 1 due to chaining. This will ensure continued transfer and double buffering. Each descriptor is also configured to interrupt the CPU at its transfer completion. This will inform the CPU that one buffer is available for processing.

5 PSoC™ 6 application notes

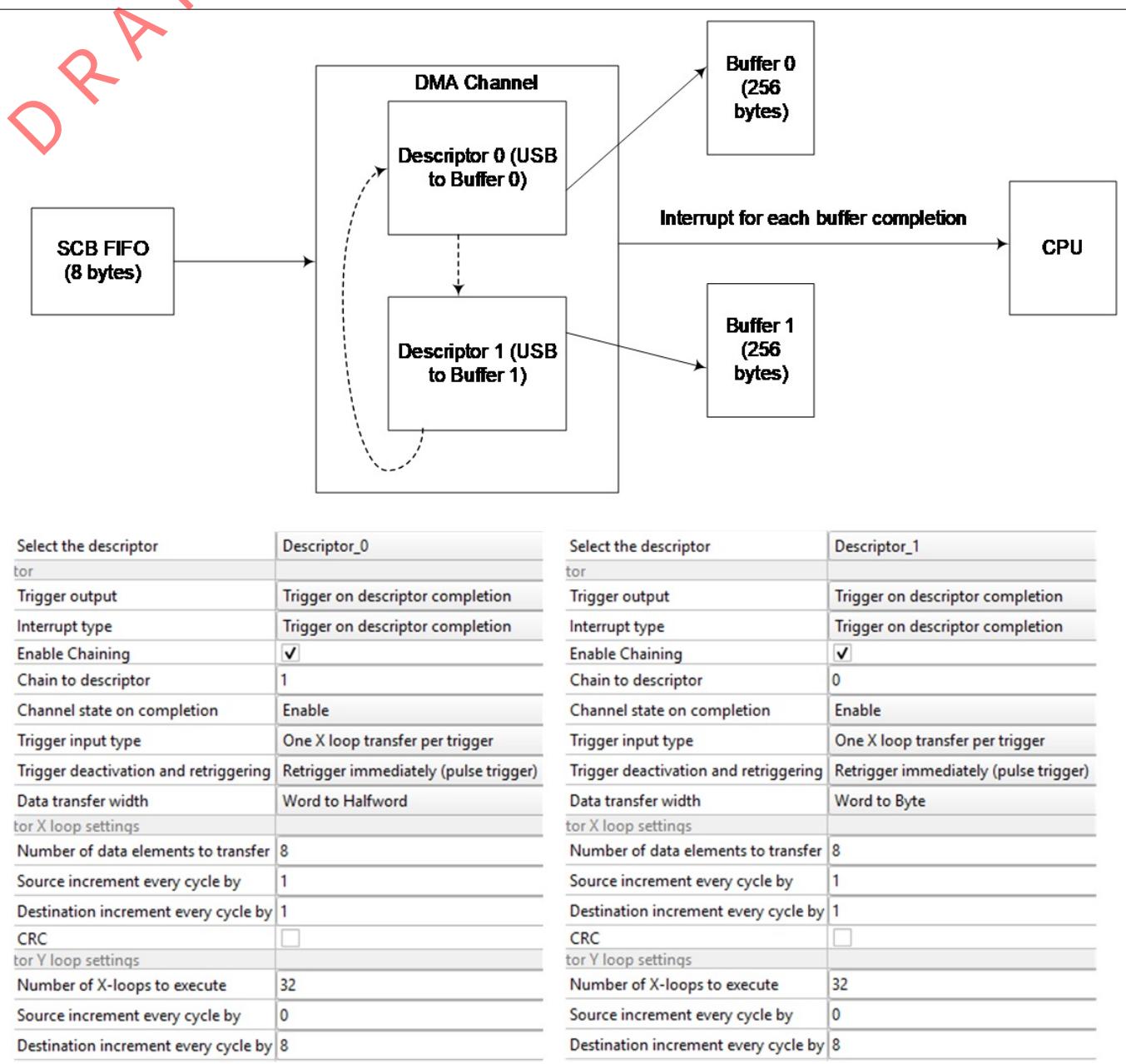


Figure 150 Double-buffering using descriptor chaining

~~DO NOT USE~~ 5 PSoC™ 6 application notes

5.4.8 Chaining DMA channels

In addition to chaining of descriptors in a single DMA channel, there are cases where it is useful to chain two DMA channels. To do this, one DMA channel's trigger output is routed to the next DMA channel's trigger input. Depending on the specific trigger multiplexer routing in a given PSoC™ 6 MCU device, only certain DMA channels will have the ability to chain. See the “Trigger Multiplexer” chapter in the Technical Reference Manual to understand the chaining restrictions of DMA channels.

A good example where DMA channels are chained is shown in [Figure 151](#). Here, the ADC input is a large analog mux. Because the number of input channels is large, it is not supported by the inbuilt multiplexer in the ADC hardware. This analog multiplexer must be implemented using an Analog Mux resource that must have routing registers modified to connect each channel. When an ADC conversion is completed, the ADC DMA channel is triggered. This moves data from the ADC result register to the memory buffer. After completion of the transfer, the ADC DMA channel triggers the MUX DMA channel. The source for this channel is a set of memory locations with preset routing values for the routing registers. Whenever this DMA channel is triggered, it transfers the new routing values to the Analog Mux registers, which effectively switch the channel.

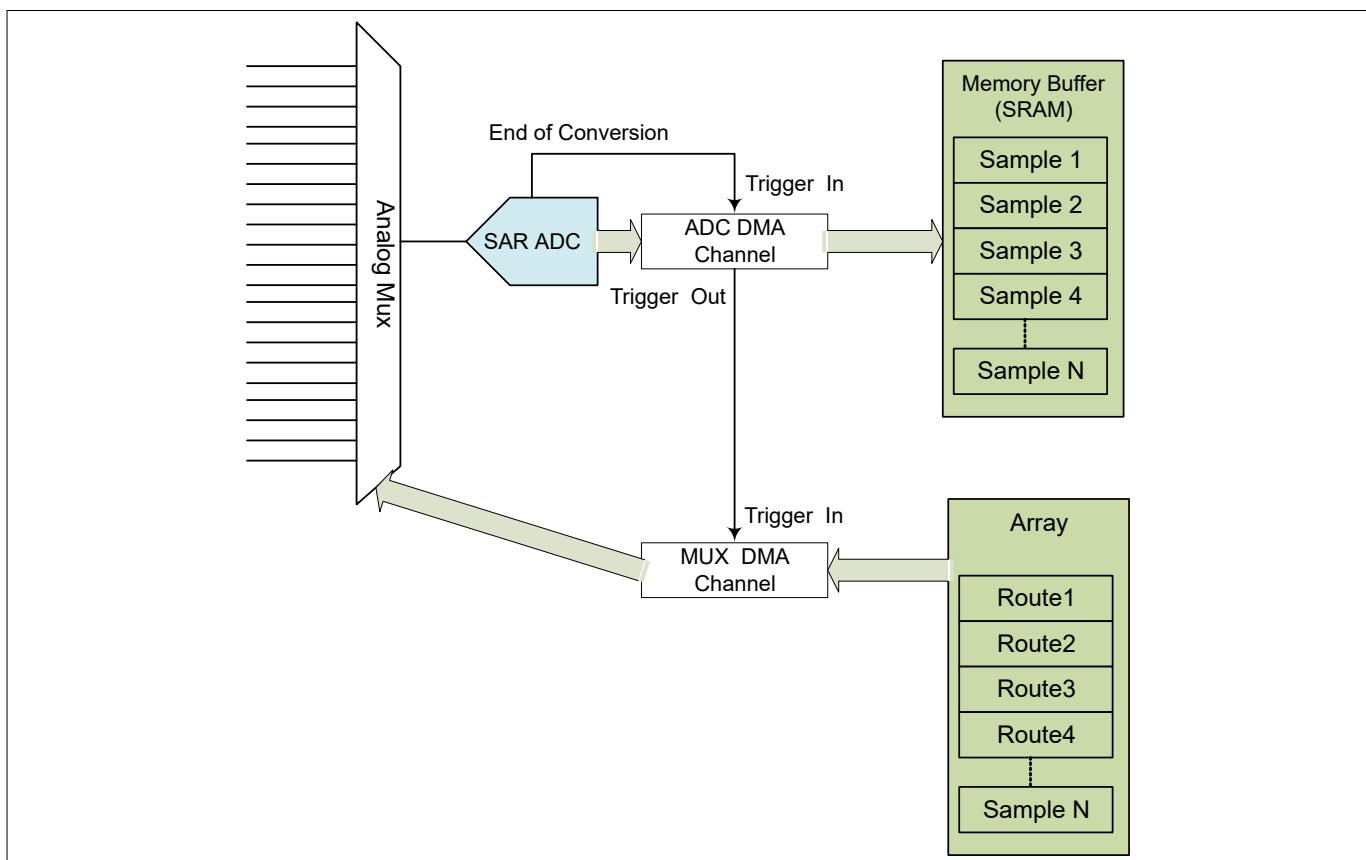


Figure 151 DMA channel chaining example

~~5 PSoC™ 6 application notes~~

~~5.4.9~~ Differences between DMA (DW) and DMAC

The main difference between DMA (DW) and DMAC relates to their usage. DMA (DW) is meant as a small data size, transactional DMA transfer. It would typically be used for transferring bytes between peripherals like from ADC to RAM. Using DMA (DW) for large transaction is expensive on a system due to its relatively low performance.

DMAC is substantially more efficient than DMA (DW) in transferring large blocks of data and should be used whenever there is a need to transfer large amounts of data in memory. DMAC focuses on achieving high memory bandwidth for a small number of channels.

There are some architectural differences in DMAC that facilitate a higher performance.

DMAC has a dedicated channel logic with dedicated channel state register for each channel. This helps retain the channel state through arbitration or preemption. Therefore, DMAC will incur fewer cycles while arbitrating between channels. This is particularly beneficial when working with multiple channels transferring large blocks of data.

DMAC has a special transfer mode called the “memory transfer mode”, which is specifically designed for fast memory-to-memory transfers. The mode is a specialized 1D transfer, where the source and destination increments are fixed to 1.

DMAC also has a 12-byte FIFO which is used for prefetch. The prefetch works by prefetching the source data as soon as the channel is enabled. When the channel is triggered, the prefetched data is simply transmitted to the destination. This shortens the initial delay of data transmission. However, this feature should be used with care to ensure that data synchronization is not violated. If the source data changes between transfers, using the prefetch buffer is not recommended because it can lead to data synchronization issues.

5 PSoC™ 6 application notes

~~DRAFT~~ 5.4.10 DMA transfer performance

The DMA block runs on the slow clock of PSoC™ 6 MCU. This section will analyze and detail how to determine the performance of a DMA transfer based on its settings. All data is based on slow clock cycles. The performance calculation numbers for all interactions over the bus presume that there is no delay due to bus arbitration. If there is such a delay, that should be added to the calculation.

5.4.10.1 Elements of a transfer

A transfer can be split into multiple operations as shown in [Table 14](#) with the corresponding cycles needed for their execution. Each transaction is initiated by a trigger, which goes through trigger synchronization circuit and takes up two cycles. These two cycles will be consumed whenever there is a trigger event being used. Loading the channel configuration takes three cycles. The loading of descriptors and the next pointer is done over the bus and therefore depends on the arbitration on the bus. Similarly, data transfers are also over the bus and therefore dependent on the arbitration happening on the bus. After the descriptor and next pointer are loaded, the DMA engine starts data transfer from the source address to the destination address. The data transfer consumes three cycles per data element.

Table 14 Operations in a transfer

Operation	Cycles (Slow Clock Cycles)
Trigger Synchronization and Priority decoding	3
Start state machine and load channel config	3
Load descriptors	4 for single transfer 5 for 1D transfer 6 for 2D transfer
Load next pointer	1
Moving data from source to destination	3

A simple single transfer will take 14 cycles. However, with the next trigger, it still must go through all 14 cycles to set up the transfer. Therefore, single transfers are not very efficient for moving multiple elements of data.

5 PSoC™ 6 application notes

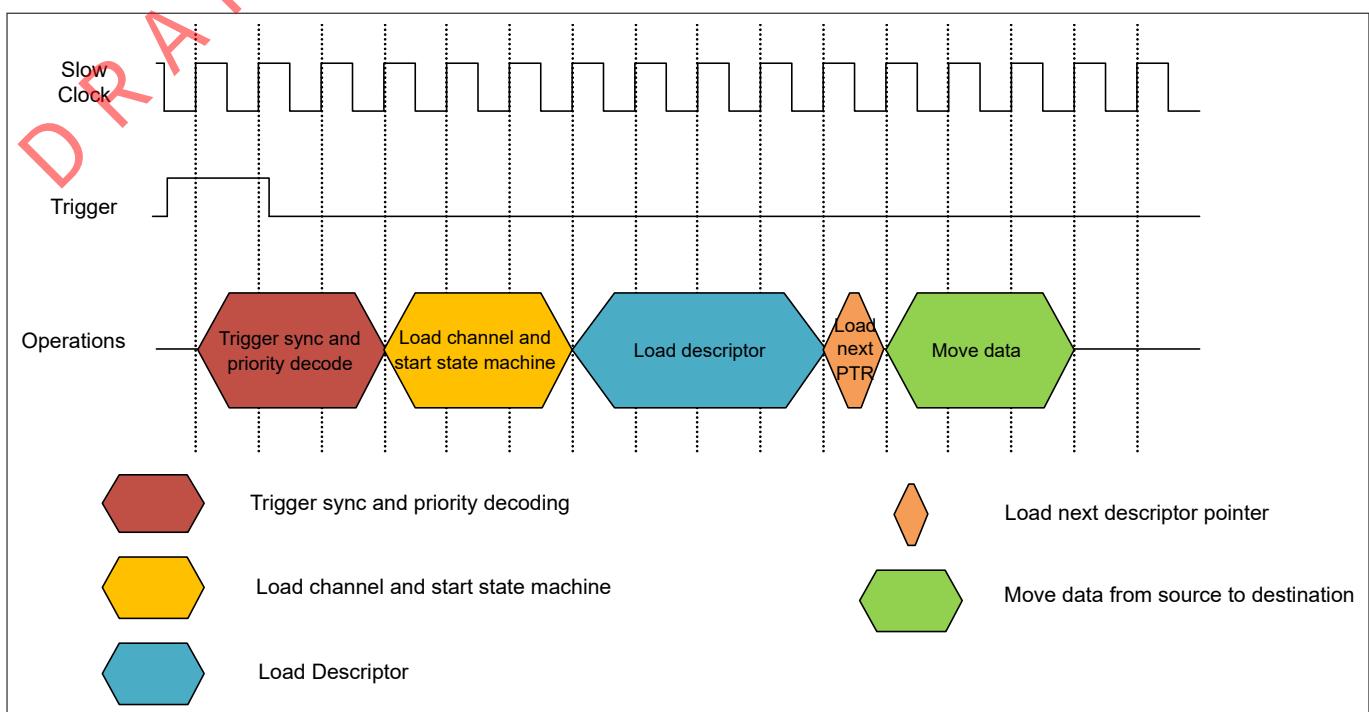


Figure 152 Timing diagram for a single transfer on the DMATiming diagram for N-byte transfer in 1D and 2D modes

1D and 2D transfers are more efficient in this regard even though they incur extra cycles to fetch additional words in the descriptor. However, these are incurred only on the first transaction; every subsequent transaction will only consume three cycles. This is shown in [Figure 153](#). Note an additional cycle is required to fetch the descriptor in a 2D transfer, but 2D transfer supports more transfer sizes compared to 1D transfer. Therefore, for larger data transfers, 2D transfer is more efficient.

5 PSoC™ 6 application notes

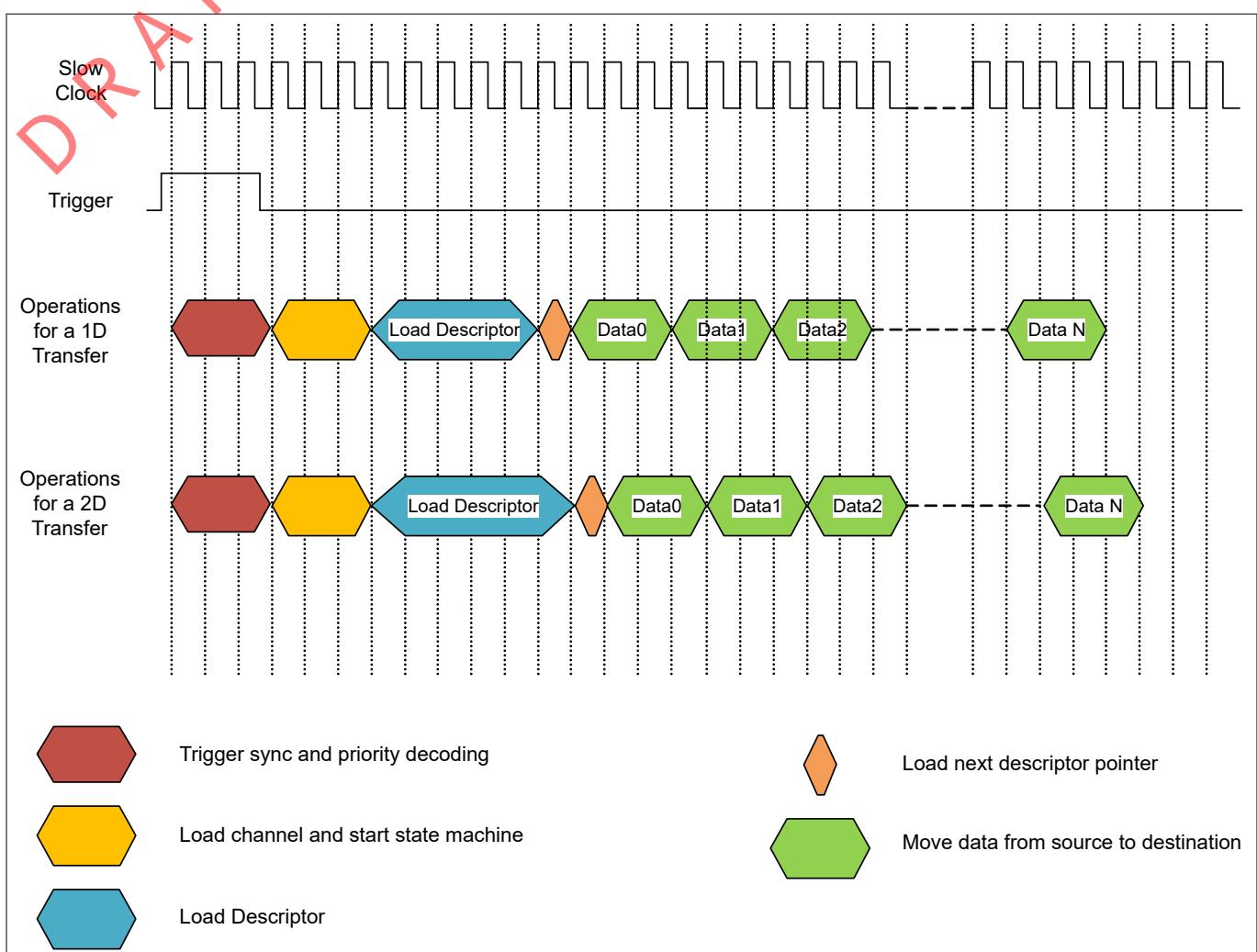


Figure 153 Timing diagram for N-byte transfer in 1D and 2D modes

To understand this better, compare the three transfer modes ([Table 15](#)) if transferring 1024 words of data. Single transfer is configured to run 1024 times. 1D transfer has a limit of 256 words per X loop. So, it would run the descriptor four times. 2D transfer will have an X loop of 256 words and a Y loop of 4 X loops.

2D transfers can move the largest amount of data at the highest throughput on a single trigger, while single transfers are the least efficient, because they treat every transaction as a new transfer and go through all the cycles of setting up the channel. Therefore, single transfers are only useful in making single element transfers on a trigger. Using single transfers to implement bulk transfers is not recommended.

Table 15 Comparing single, 1D, and 2D transfers

(Number of cycles)	Single transfer (one transfer per trigger)	1D transfer (entire descriptor per trigger)	2D transfer(entire descriptor per trigger)
First element transferred (1 time) in number of slow clock cycles	14	15	16
First element of each X loop (3 times) in number of slow clock cycles	14	15	3

(table continues...)

5 PSoC™ 6 application notes

DRAFT
Table 15 (continued) Comparing single, 1D, and 2D transfers

(Number of cycles)	Single transfer (one transfer per trigger)	1D transfer (entire descriptor per trigger)	2D transfer(entire descriptor per trigger)
All other transfers in number of slow clock cycles (1020 times)	14	3	3
Total (1024 bytes) in number of slow clock cycles	14336	3120	3085
Throughput (MB/s) Slow clock= 50 MHz Transfer width= 32-bit	14.28	65.64	66.68

5.4.10.2 DMA (DW) and DMAC: trigger schemes and performance

Note that whenever a trigger is to be processed, the DMA (DW) incurs the entire 14-16 cycles (depending on the transfer mode). This is because every time the DMA hardware block is in a state where it needs to wait for a trigger, it may switch channels to service other pending channels. So, when the next trigger is initiated, DMA (DW) goes through the additional cycles needed to start the transfer. Thus, performance is degraded where triggers are processed more often. [Table 16](#) compares trigger schemes in a 2D transfer for a DMA (DW) that is set up to transfer 1024 bytes.

Table 16 Performance of trigger schemes in DMA (DW)

(Number of cycles)	2D transfer(one transfer per trigger)	2D transfer(one X loop per trigger)	2D transfer(entire descriptor per trigger)
First element transferred	16	16	16
First element of each X loop	16	16	3
All other transfers	16	3	3
Total (slow clock cycles) (1024 bytes)	16384	3124	3085

On DMAC, the DMA transfer engine logic is replicated per channel with channel-level memory to hold the state while switching channels. For every trigger, the DMAC does not have to fetch all channel and descriptor information. The only extra cycles incurred are for trigger sync and priority decoding. This means that DMAC will perform better in a transfer that is triggered multiple times. [Figure 154](#) shows this comparison of performance between DMAC and DMA (DW) using a timing diagram.

5 PSoC™ 6 application notes

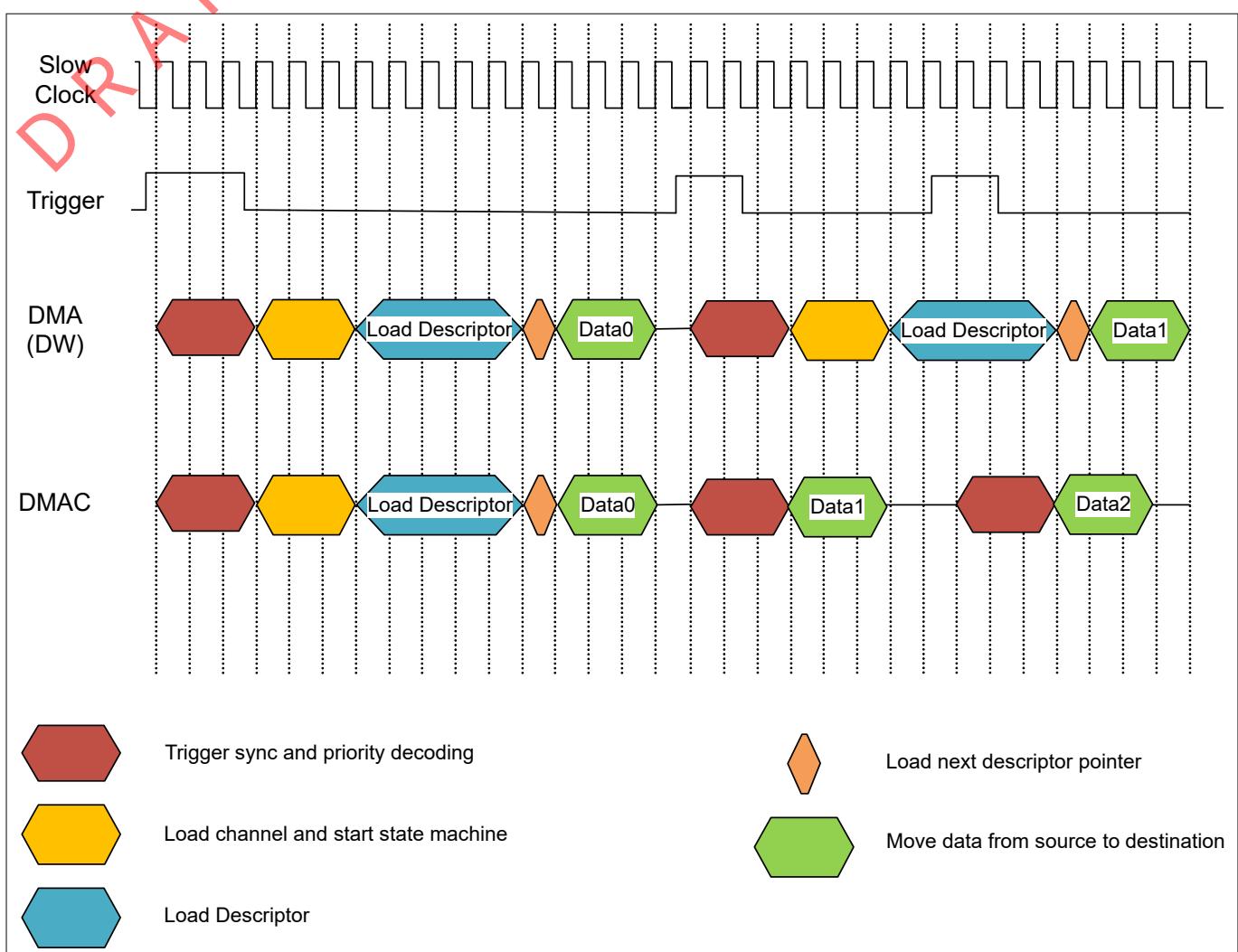


Figure 154 Performance comparison of DW and DMAC for a triggered mode of transfer

Also note that, when a transaction is waiting for a trigger, it could get preempted by another channel. This can lead to additional delays waiting for the preempting channel to relieve the DMA hardware block. This is especially an issue when dealing with a system with multiple DMA channels which have competing priorities. From this performance comparison it is clear that

1. Transfers are more efficient when transactions are done in bulk without waiting for triggers. Triggering requirement adds delay
2. Using 2D transfers improves performance when the entire transfer is completed in a single trigger
3. Waiting for a trigger can seriously drop performance due to the possibility of arbitration with other channels in the DMA hardware block
4. If there is a need to use a trigger for individual transfer elements, using the DMAC will give you a better performance than the DMA (DW)

5.4.10.3 Preemption and its impact on performance

The choice of making a DMA (DW) channel as preemptable affects its performance. This is because every time a channel is preempted, the following happen:

5 PSoC™ 6 application notes

- The channel is in pending state for as long as a higher-priority channel is running
- On resumption, the channel descriptor must be fetched again, costing additional cycles for every resume. So, if there are large number of higher-priority channels, making a low-priority channel preemptible can have adverse effects on its throughput

During preemption, the throughput of a low-priority DMA (DW) channel is affected by the delay caused by the high-priority channel and the extra cycles incurred for each time it was preempted. If the low-priority channel was transferring large blocks of data, this can incur multiple instances of preemption and the effect of the extra cycles incurred start hurting the throughput.

This impact is limited for DMAC channels because they do not need to go through the entire descriptor fetch cycle when resuming after a preemption. This means that the throughput of a low-priority DMAC channel is only affected by the delay caused by the high-priority channel. When transferring large blocks of data through a low-priority preemptable channel, it is better to use DMAC for a better throughput.

The performance comparison in a preemption scenario is shown in [Figure 155](#).

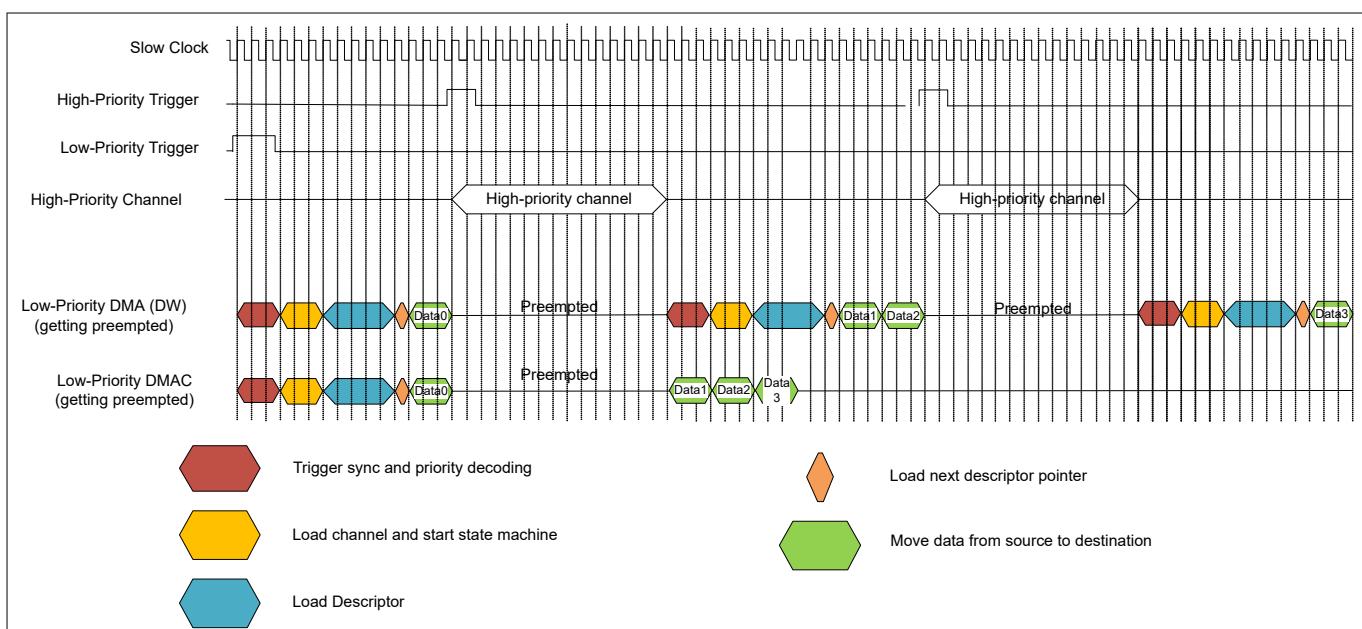


Figure 155 Comparing performance of DMAC and DMA(DW) in a preemption scenario

On the other hand, if there is a low-priority channel that is transferring a large amount of data, not making it preemptable can starve other high-priority channels for too long. Only channels whose data transfers are not time-critical can be made preemptable.

Sometimes, you can also distribute channels across multiple DW blocks to avoid conditions of preemption and deal with contention at the bus arbitration level.

5.4.10.4 Bus arbitration and its impact

There are multiple bus masters in the PSoC™ 6 MCU device. A DMA channel may have the highest priority in its DMA hardware block, but that does not guarantee its performance on the bus when arbitrating with other bus masters. Sometimes, even a DMA channel's descriptor fetch process can be delayed due to bus arbitration by other masters. This issue can be minimized by controlling the arbitration scheme configured in PROT_SMPU_MSx_CTL[PRIO]. For more details, see the Registers TRM.

5 PSoC™ 6 application notes~~DO NOT USE~~
5.4.11 Summary

PSoC™ 6 MCU family devices offer two types of DMA blocks: DMA (DW) and DMAC. The DMA (DW) offers a lot of DMA channels meant for small data size transfers, typically between peripherals and memory. The DMAC is a high-performance DMA meant for large data transfers, typically between memory locations, with maximum throughput.

5 PSoC™ 6 application notes

References

- 4

For a comprehensive list of PSoC™ 6 MCU resources, see [KBA223067](#) in the Infineon community.

For a comprehensive list of PSoC™ 3, PSoC™ 4, and PSoC™ 5LP resources, see [KBA86521](#) in the Infineon community.

Application Notes

- [1] [AN210781](#) – Getting Started with PSoC™ 6 MCU with Bluetooth® Low Energy Connectivity: Describes PSoC™ 6 MCU with Bluetooth® LE Connectivity devices and how to build your first PSoC™ Creator project
- [2] [AN215656](#) – PSoC™ 6 MCU: Dual-CPU System Design: Describes the dual-CPU architecture in PSoC™ 6 MCU, and shows how to build a simple dual-CPU design
- [3] [AN219434](#) – Importing PSoC™ Creator Code into an IDE for a PSoC™ 6 MCU Project: Describes how to import the code generated by PSoC™ Creator into your preferred IDE

Code Examples

- [1] [CE218553](#) – PWM Triggering a DMA Channel
- [2] [CE218552](#) – UART to Memory Buffer Using DMA
- [3] [CE225786](#) – PSoC™ 6 MCU USB Audio Recorder
- [4] [CE222221](#) – PSoC 6 MCU Voice Recorder
- [5] [CE220762](#) – PSoC 6 MCU PDM to I2S Example

Device Documentation

- [1] [PSoC 6 MCU: PSoC 63 with BLE Datasheet](#)
- [2] [PSoC 6 MCU: PSoC 63 with BLE Architecture Technical Reference Manual](#)

Development Kit Documentation

- [1] [CY8CKIT -062- BLE PSoC 6 BLE Pioneer Kit](#)

Tool Documentation

- [1] [PSoC™ Creator](#): Look in the Downloads tab for Quick Start and User Guides
- [2] [Peripheral Driver Library \(PDL\)](#): Installed by PSoC™ Creator 4.2. Look in the <PDL install folder>/doc for the User Guide and the API Reference
- [3] [ModusToolbox™ Software](#): Look in the Quick Panel under the heading Documentation. Alternately look in the install directory for ModusToolbox™ <ModusToolbox install folder>/doc
- [4] WICED SDK with PSoC™ 6 Support: Installed with ModusToolbox™ Software

5 PSoC™ 6 application notes**5.4.12 Revision history**

Document version	Date of release	Description of changes
**	2020-05-29	Initial release
*A	2021-02-23	Updated in Infineon template
*B	2022-07-21	Template update

5.5 AN233648 PSoC™ 6 MCU: Modify CY8CPROTO-062-4343W board to work with an external flash memory**About this document**

- 5

Scope and purpose

This application note describes steps to enable a CY8CPROTO-062-4343W PSoC™ 6 Wi-Fi Bluetooth® prototyping kit to work with an external flash memory by modifying the board hardware and software. By replacing the attached flash with the external connection and adding commands into PSoC™ 6 MCU serial memory interface (SMIF), this board can be turned into a memory evaluation board for any external flash.

Intended audience

This application note targets PSoC™ 6 MCU board users who intend to further investigate operation with external flash memory devices other than the offered S25FL512S flash on the board. This method will free the limitation of the users to evaluate any flash. The flash device in the SOIC-16 package on the board may be replaced freely; the external flash outside the board with any type of package can be tested.

5 PSoC™ 6 application notes

5.5.1 Introduction

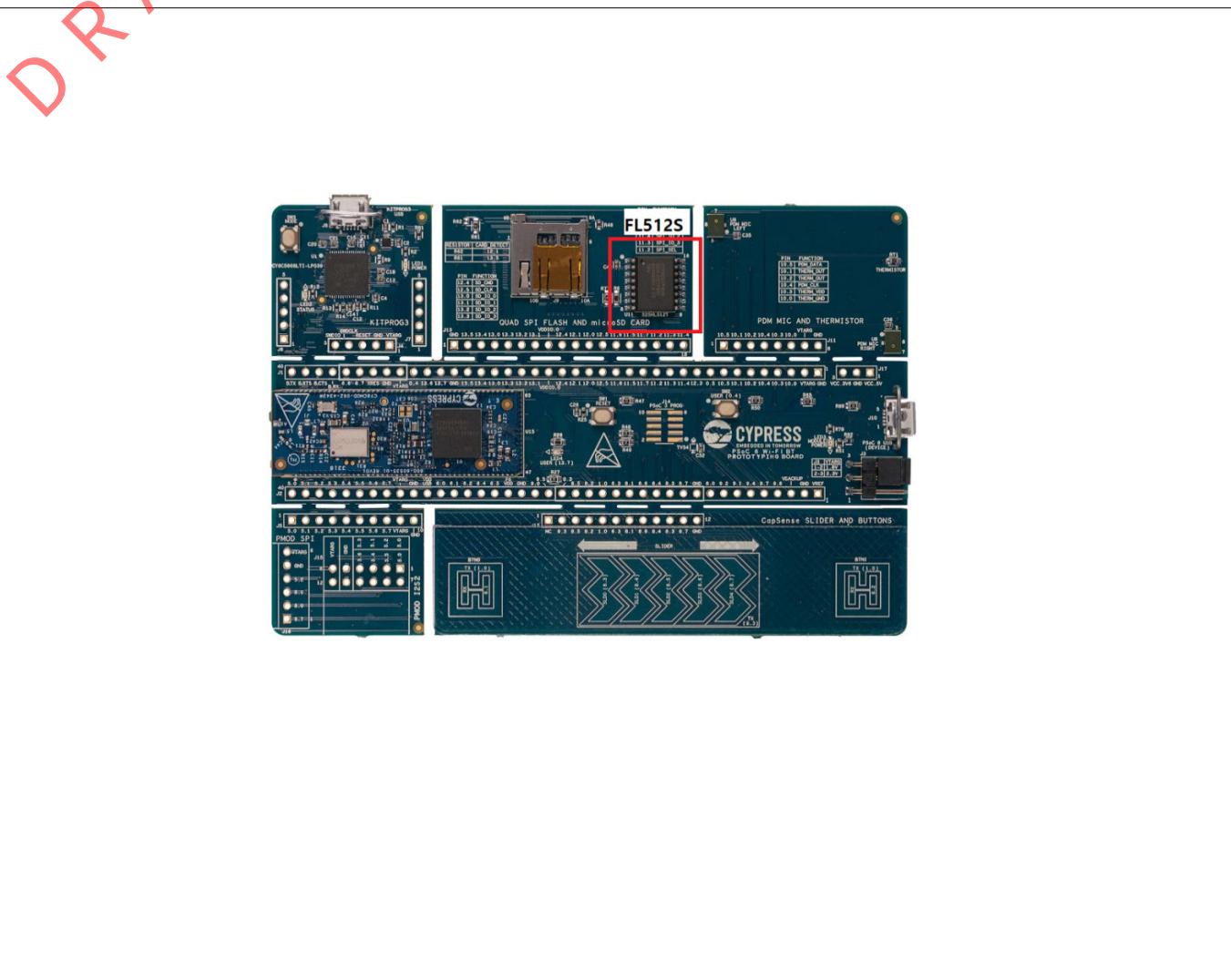


Figure 156 CY8CPROTO-062-4343W PSoC™ 6 Wi-Fi Bluetooth® prototyping board

The PSoC™ 6 Wi-Fi Bluetooth® prototyping kit (CY8CPROTO-062-4343W) is provided with the S25FL512S 512-Mb SPI NOR flash using SMIF. SMIF is an SPI-based communication interface for interfacing external devices to a PSoC™ MCU device. However, as this PSoC™ 6 MCU board comes with external flash attached to the board. Because there are several commands to evaluate all flash operations, there are limitations on evaluating flash memories fully. This application note describes a method to modify this PSoC™ 6 MCU board hardware and software for customizing the board into a general evaluation tool for flash memory devices.

5 PSoC™ 6 application notes

5.5.2 Procedures

Both hardware and software modifications are required for this application.

5.5.2.1 Modifying the hardware

5.5.2.1.1 Materials required

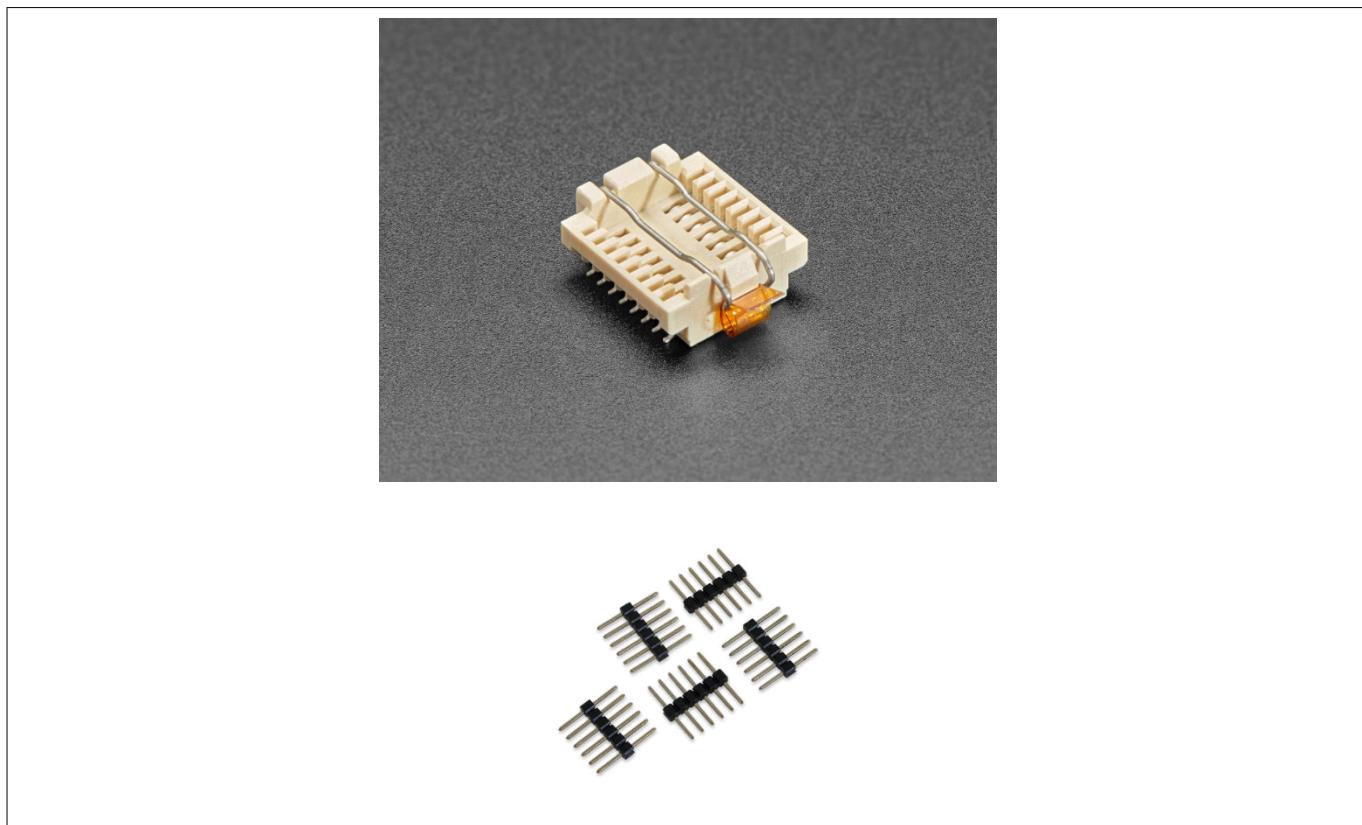


Figure 157 SMT socket – wide SOIC-16, pin header

- CY8CPROTO-062-4343W PSoC™ 6 W-Fi Bluetooth® prototyping kit board
- SMT socket-wide SOIC-16
- 8-pin headers
- Cutter
- Soldering machine

5.5.2.1.2 Customize the board

Prepare the board and SMT socket-wide SOIC-16, and 8-pin headers. You may also need a cutter to remove the existing flash device and a soldering machine to solder the socket and pin headers.

1. Remove the attached flash with the cutter. Gently cut off the flash lead frames and then remove the leftovers with solders. Be careful not to strip off the PCB pattern
2. Fit the SMT socket-wide SOIC-16 into the PCB footprint. The existing flash on the board is in SOIC 16 pin package, so it is compatible with the new socket. Solder the socket along the footprint
3. Solder the pin headers to GND, VDDIO, pins 11.6, 11.5, 11.7, 11.2, 11.3, and 11.4 pin holes. See [Table 17](#) and [Figure 158](#)
4. Make sure that all pins are connected

~~DRAFT~~
5 PSoC™ 6 application notes

Table 17 CY8CPROTO-062-4343W pinout

Pin	Flash	Pin	Flash
GND	VSS	P11.7	SCK
VDDIO_O	VCC	P11.2	#CS
P11.6	SI/IO0	P11.3	HOLD/IO3
P11.5	SO/IO1	P11.4	WP/IO2

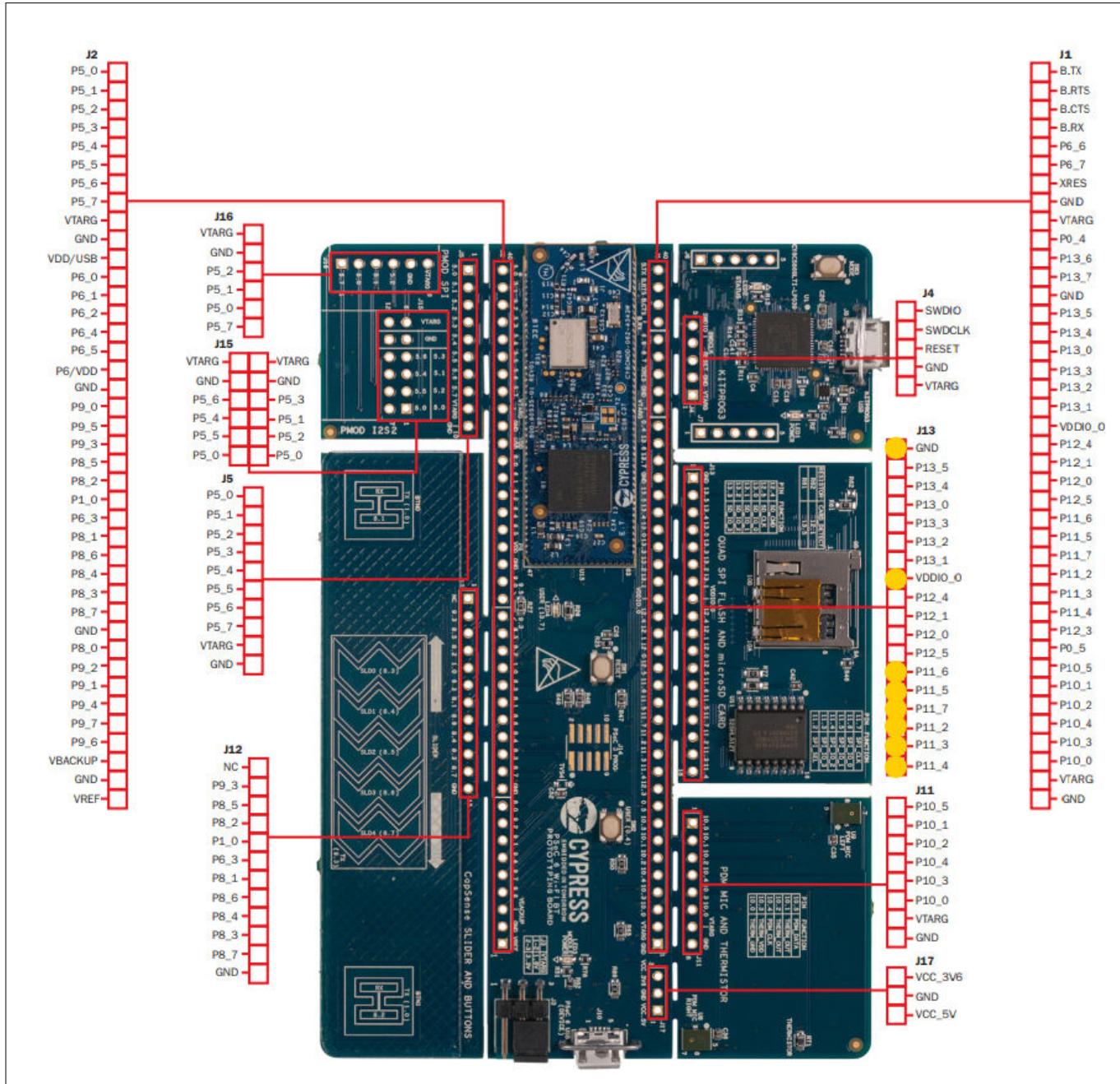


Figure 158 Board pinout

Test the flash devices as follows:

5 PSoC™ 6 application notes

- ~~DRAFT~~
- To test any flash device in SOIC-16 package, replace the flash chip in the socket as shown in [Figure 160](#)
 - To test flash devices in other package types or to test the flash device that is attached to another board like shown in [Figure 161](#), use the pin header connector. In this case, keep the socket empty because the socket flash signal and external signal through pin header can be overlapped

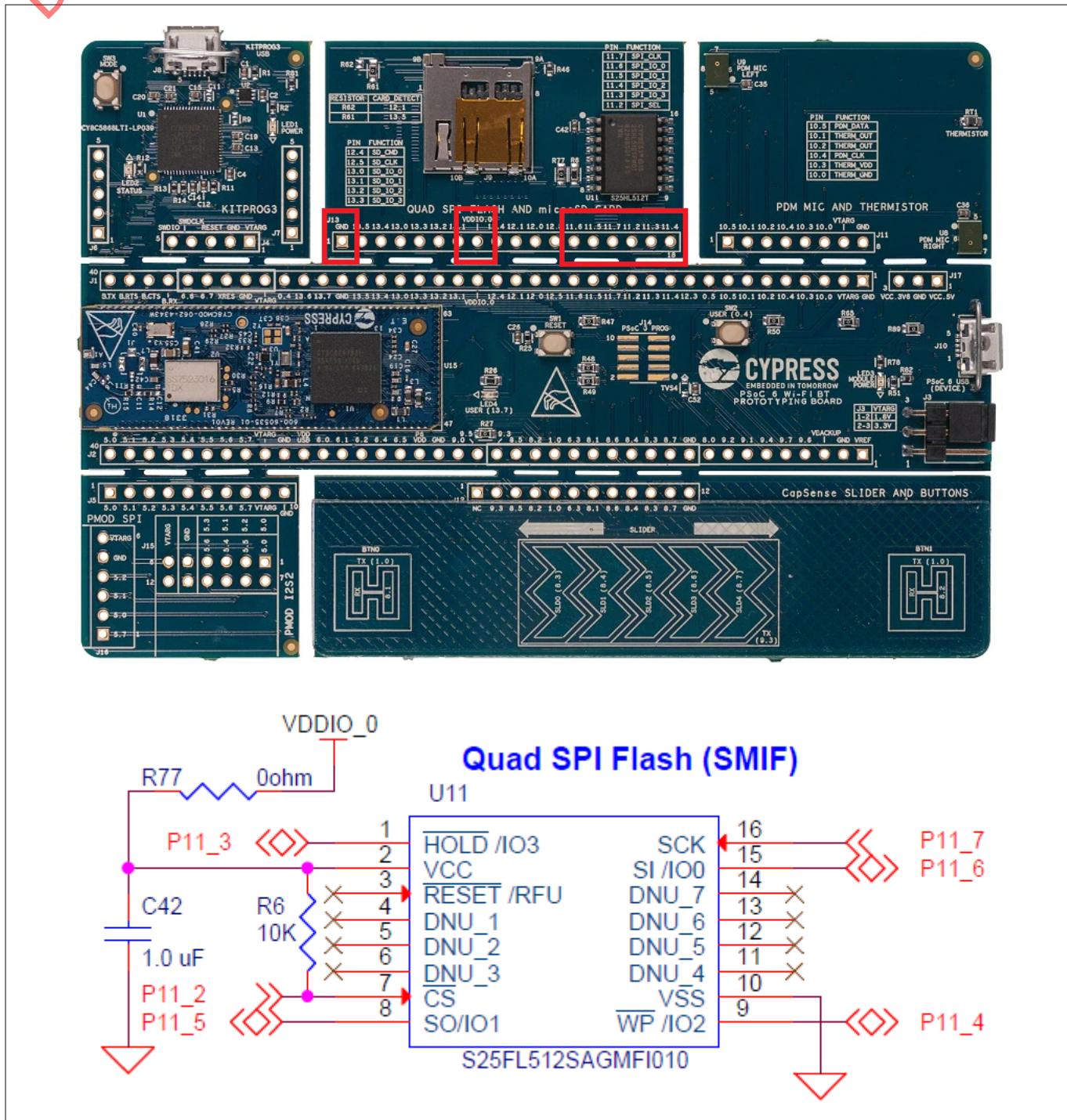


Figure 159 CY8CPROTO-062-4343W external connections

[Figure 160](#) and [Figure 161](#) show the boards with the hardware modifications:

- [Figure 160](#) shows the flash device in the socket being tested
- [Figure 161](#) shows an external flash device on another board being tested. Note that the jumpers are connected to the pin header while the socket is empty

5 PSoC™ 6 application notes



Figure 160 PSoC™ 6 MCU board with the socket attached

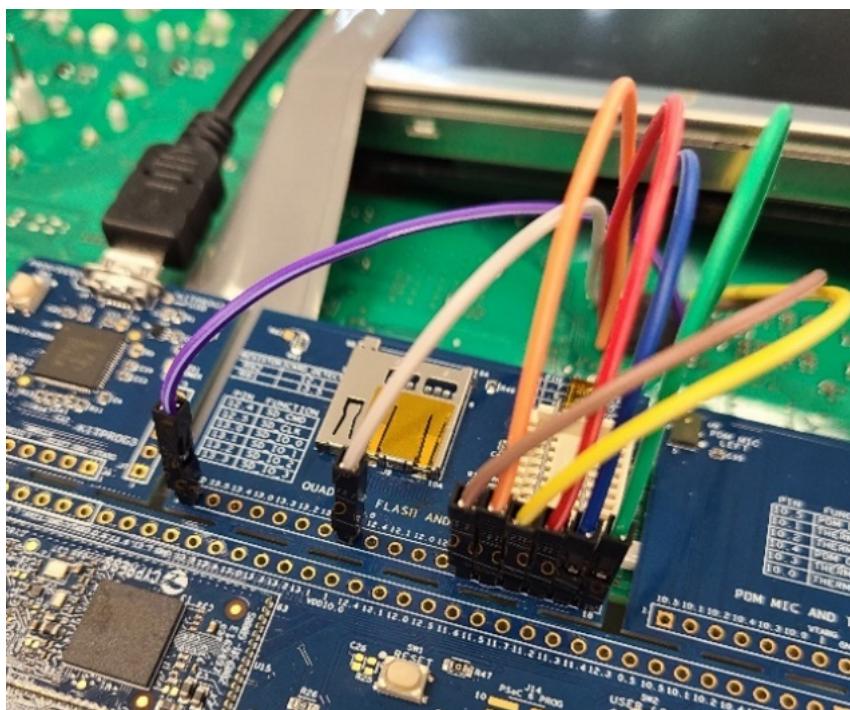


Figure 161 External flash memory connected with a pin header

5 PSoC™ 6 application notes**5.5.2.2 Modifying the software****5.5.2.2.1 ModusToolbox™ software**

PSoC™ 6 MCU boards use Eclipse IDE for ModusToolbox™ and C language for programming.



Download the software from https://www.cypress.com/products/modustoolbox#tabs-0-bottom_side-6.

5.5.2.2.2 Serial memory interface (SMIF)

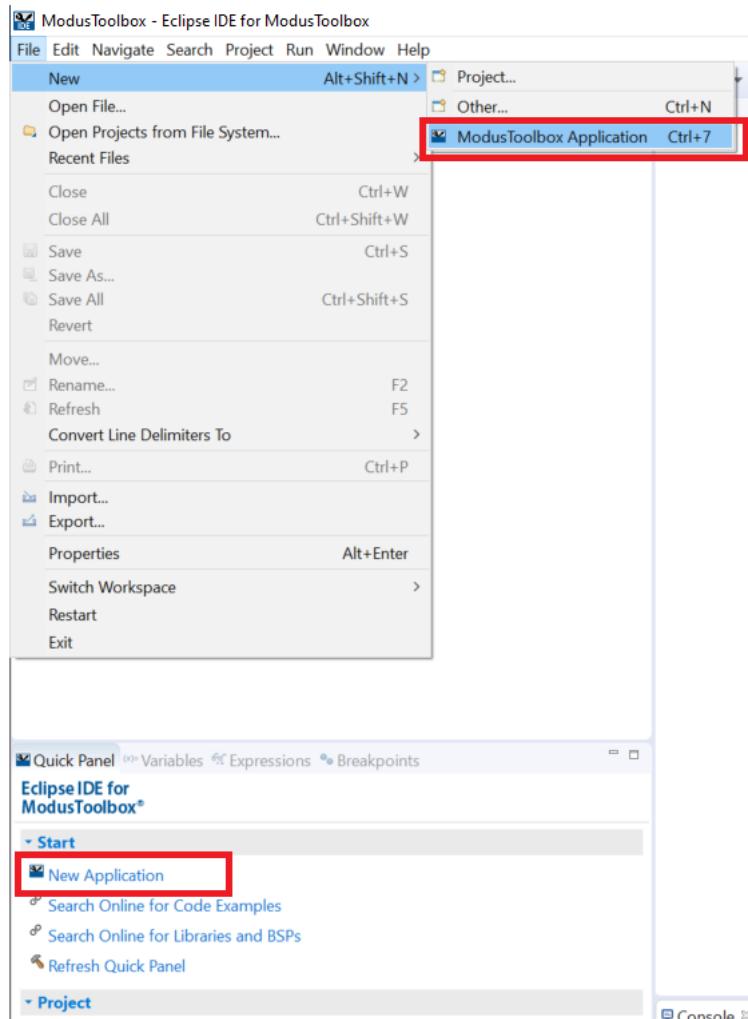
The SMIF Component implements an SPI-based communication hardware block for interfacing external memory devices with the PSoC™ 6 MCU device. The firmware uses the source code (cy_smif_memconfig.c and cy_smif_memconfig.h files) generated from the SMIF configurator. This source code defines the data structures that hold the memory configuration.

5 PSoC™ 6 application notes

5.5.2.2.3 Import ModusToolbox™ QSPI flash code example

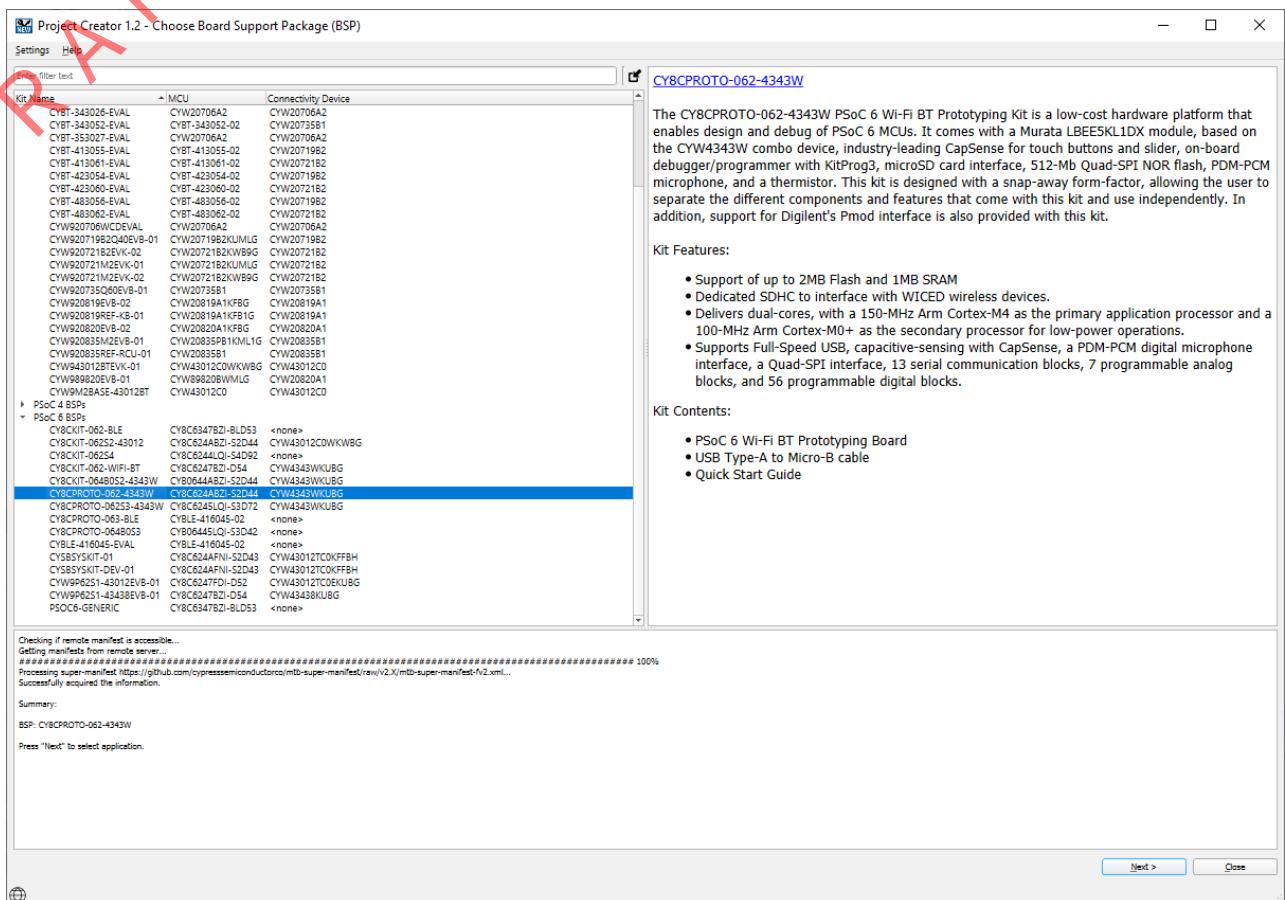
1. On Eclipse IDE for ModusToolbox™ software, import example project into your project. Do one of the following:

- Select **File > New > ModusToolbox™ Application**
- Click **New Application** on the **Start tab** on the **Quick Panel**



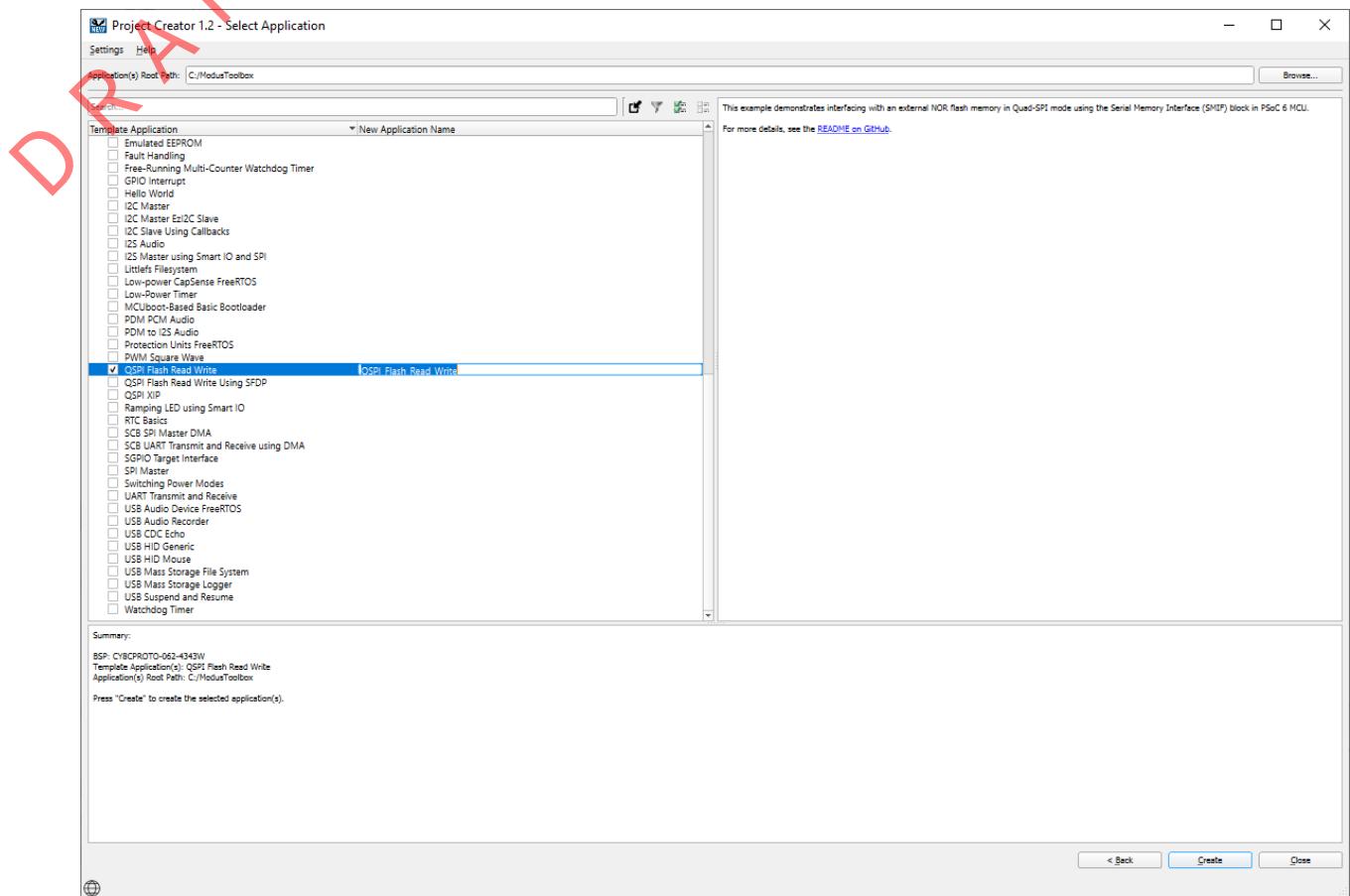
2. Choose the board support package (BSP) for the CY8CPROTO-062-4343W board and then click **Next**

5 PSoC™ 6 application notes

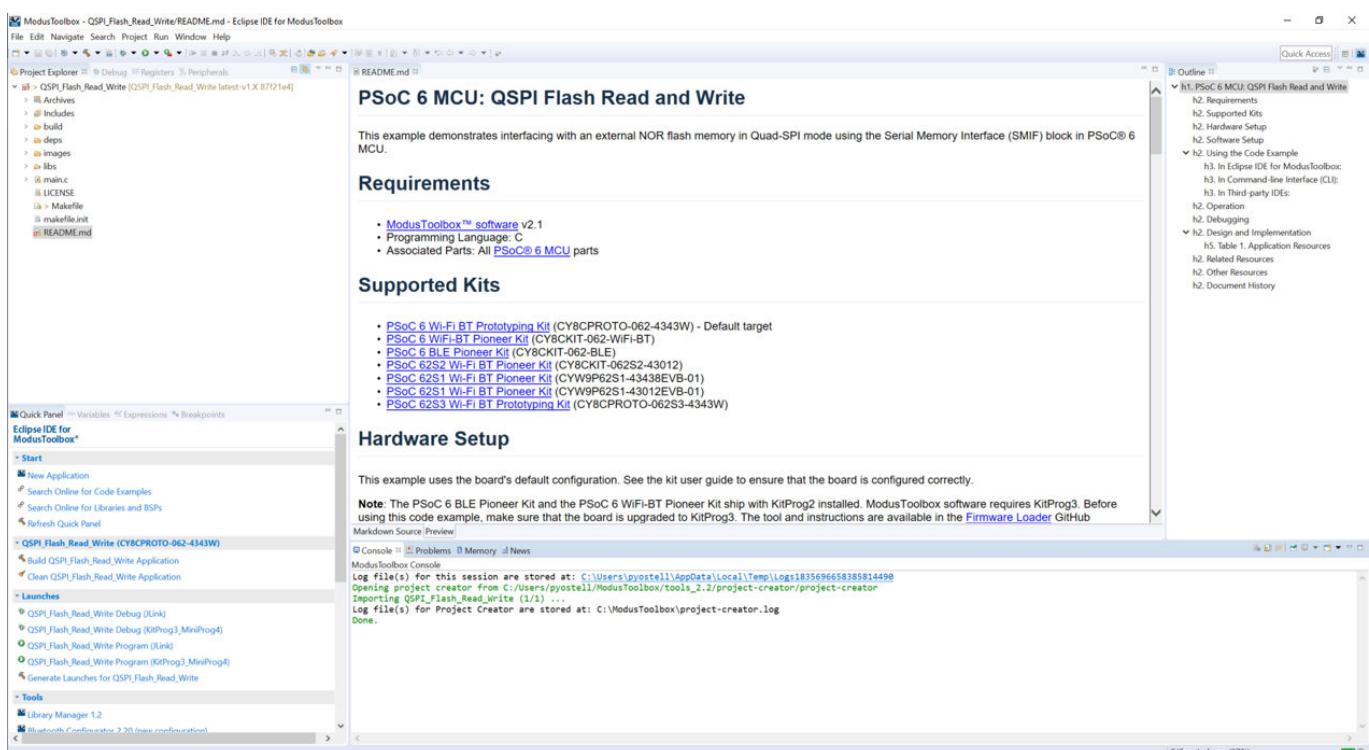


- On the Select Application window, select **QSPI Flash Read Write**. Optionally, change the application name

5 PSoC™ 6 application notes



The following screen appears when the selected example is successfully imported:



5 PSoC™ 6 application notes~~DRAFT~~
5.5.2.2.4 Determine the serial flash operation structure

Note that the application already includes supports for some flash operation commands for QSPI Flash Read and Write operation. The `cy_smif_memslot.h` file includes code to support the WRR(01h), WRDI(04h), RDSR1(05h), WREN(06h), 4QPP(34h), RDCR(35h), BE(60h), 4SE(DCh), and 4QIOR(ECh) commands. However, these commands may not be sufficient to test all flash operations.

5 PSoC™ 6 application notes

Code Listing 1: Built-in functions in cycfh_qspi_memslot.c

DRAFT

```
typedef struct
{
    /*This specifies the number of address bytes used by the memory slave device, valid values 1-4 */
    uint32_t numOfAddrBytes;
    /*The memory size: For densities of 2 gigabits or less - the size in bytes; For densities 4 gigabits and above - bit-31 is set to 1b to define that this memory is 4 gigabits and above; and other 30:0 bits define N where the density is computed as 2^N bytes. For example, 0x80000021 corresponds to 2^30 = 1 gigabyte.*/
    uint32_t memSize;
    /*This specifies the Read command */
    cy_stc_smif_mem_cmd_t* readCmd;
    /*This specifies the Write Enable command */
    cy_stc_smif_mem_cmd_t* writeEnCmd;
    /*This specifies the Write Disable command */
    cy_stc_smif_mem_cmd_t* writeDisCmd;
    /*This specifies the Erase command */
    cy_stc_smif_mem_cmd_t* eraseCmd;
    /*This specifies the sector size of each Erase */
    uint32_t eraseSize;
    /*This specifies the Chip Erase command */
    cy_stc_smif_mem_cmd_t* chipEraseCmd;
    /*This specifies the Program command */
    cy_stc_smif_mem_cmd_t* programCmd;
    /*This specifies the page size for programming */
    uint32_t programSize;
    /*This specifies the command to read the WIP-containing status register */
    cy_stc_smif_mem_cmd_t* readStsRegWipCmd;
    /*This specifies the command to read the QE-containing status register */
    cy_stc_smif_mem_cmd_t* readStsRegQeCmd;
    /*This specifies the command to write into the QE-containing status register */
    cy_stc_smif_mem_cmd_t* writeStsRegQeCmd;
    /*This specifies the read SFDP command */
    cy_stc_smif_mem_cmd_t* readSfdpCmd;
    /*This specifies the Read ID command */
    cy_stc_smif_mem_cmd_t* readIDCmd;
    /* The Busy mask for the status registers */
    uint32_t stsRegBusyMask;
    /*The QE mask for the status registers */
    uint32_t stsRegQuadEnableMask;
    /*Max time for erase type 1 cycle time in ms*/
    uint32_t eraseTime;
    /*Max time for chip erase cycle time in ms */
    uint32_t chipEraseTime;
    /*Max time for page program cycle time in us */
    uint32_t programTime;
    /*This specifies the number of regions for memory with hybrid sectors */
    uint32_t hybridRegionCount;
    /*This specifies data for memory with hybrid sectors */
}
```

5 PSoC™ 6 application notes

```
cy_stc_smif_hybrid_region_info_t** hybridRegionInfo;
} cy_stc_smif_mem_device_cfg_t;
```

This section provides steps to add additional operation commands other than the commands provided by the example code. The flash commands vary from the products and the interface—see the datasheet for each flash for supported command sets.

For example, a brief list of supported commands for the S25FL512S device is shown in [Table 18](#). See the datasheet for detailed descriptions for each command.

The example code provided commands WRR(01h), WRDI(04h), RDSR1(05h), WREN(06h), 4QPP(34h), RDCR(35h), BE(60h), 4SE(DCh), 4QIOR(ECh) are shown in **Bold**.

Table 18 Command lists

Instruction (in HEX)	Command name	Instruction (in HEX)	Command name
01	WRR	6C	4QOR
02	PP	75	ERSP
03	READ	7A	ERRS
04	WRDI	85	PGSP
05	RDSR1	8A	PGRS
06	WREN	90	READ_ID
07	RDSR2	9F	RDID
0B	FAST_READ	A3	MPM
0C	4FAST_READ	A6	PLBWR
0D	DDRFR	A7	PLBRD
0E	4DDRFR	AB	RES
12	4PP	B9	BRAC
13	4READ	BB	DIOR
14	ABRD	BC	4DIOR
15	ABWR	BD	DDRDIOR
16	BRRD	BE	4DDRDIOR
17	BRWR	C7	BE
18	ECCRD	D8	SE
20	P4E	DC	4SE
21	4P4E	E0	DYBRD
2B	ASPRD	E1	DYBWR
2F	ASPP	E2	PPBRD
30	CLSR	E3	PPBP
32	QPP	E4	PPBE
34	4QPP	E5	Reserved-E5
35	RDCR	E6	Reserved-E6

(table continues...)

5 PSoC™ 6 application notes

Table 18  **(continued) Command lists**

Instruction (in HEX)	Command name	Instruction (in HEX)	Command name
38	QPP	E7	PASSRD
3B	DOR	E8	PASSP
3C	4DOR	E9	PASSU
41	DLP RD	EB	QIOR
42	OTPP	EC	4QIOR
43	PNVDLR	ED	DDRQIOR
4A	WVDLR	EE	4DDRQIOR
4B	OTPR	F0	RESET
60	BE	FF	MBR
6B	QOR		

Use the top-down approach on the codes to determine the architecture of the serial flash memory operation. For example, consider the `cy_serial_flash_qspi_read` operation starting from `main.c` to investigate the SMIF structure.

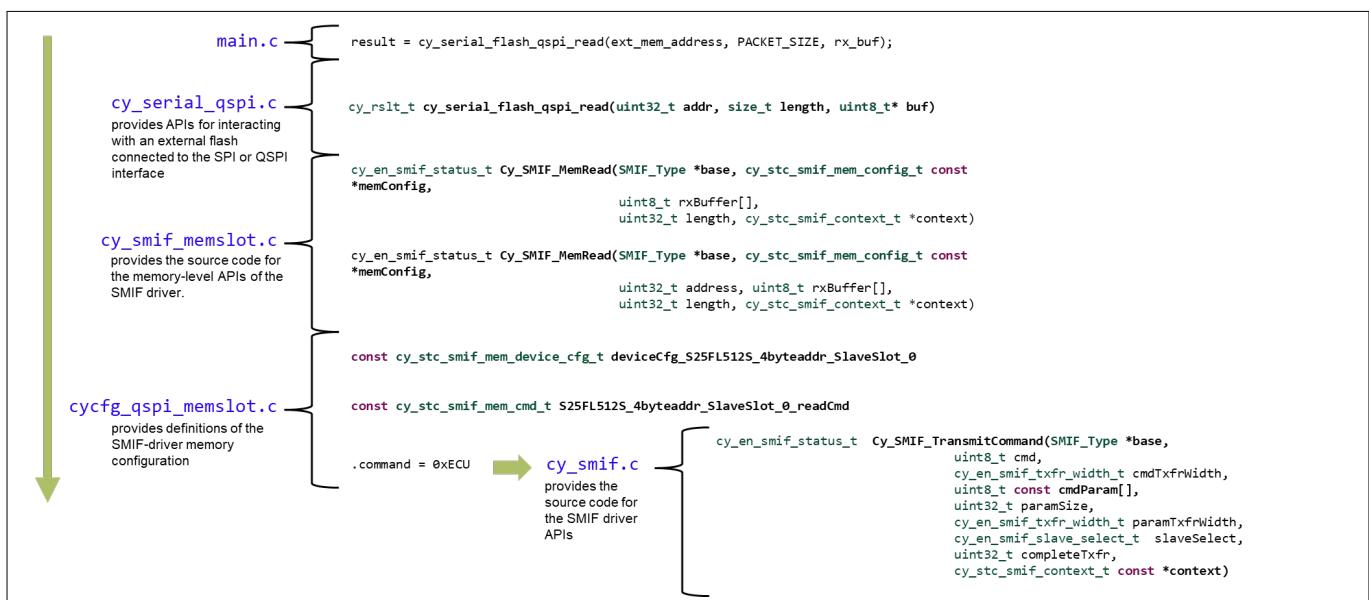


Figure 162 SMIF structure

By exploring from `main.c` and all the way down to `cy_smif.c`, you can determine that the actual operation command is delivered to `Cy_SMIF_TransmitCommand` in `cy_smif.c` with command arguments, command width, address width, mode, mode width, the number of dummy cycles, and data width. The arguments are specified in `cycfg_aspi_memslot.c` for each operation command.

5 PSoC™ 6 application notes

Code Listing 2: S25FL512S_4byteaddr_SlaveSlot_0_readCmd in cycfg_qspi_memslot.c

```

DRAFT
const cy_stc_smif_mem_cmd_t
S25FL512S_4byteaddr_SlaveSlot_0_readCmd =
{
    /* The 8-bit command. 1 x I/O read command. */
    .command = 0xECU,
    /* The width of the command transfer. */
    .cmdWidth = CY_SMIF_WIDTH_SINGLE,
    /* The width of the address transfer. */
    .addrWidth = CY_SMIF_WIDTH_QUAD,
    /* The 8-bit mode byte. This value is 0xFFFFFFFF when there is no mode present. */
    .mode = 0x01U,
    /* The width of the mode command transfer. */
    .modeWidth = CY_SMIF_WIDTH_QUAD,
    /* The number of dummy cycles. A zero value suggests no dummy cycles. */
    .dummyCycles = 4U,
    /* The width of the data transfer. */
    .dataWidth = CY_SMIF_WIDTH_QUAD
};

```

See the datasheet for the argument values in the command sequence.

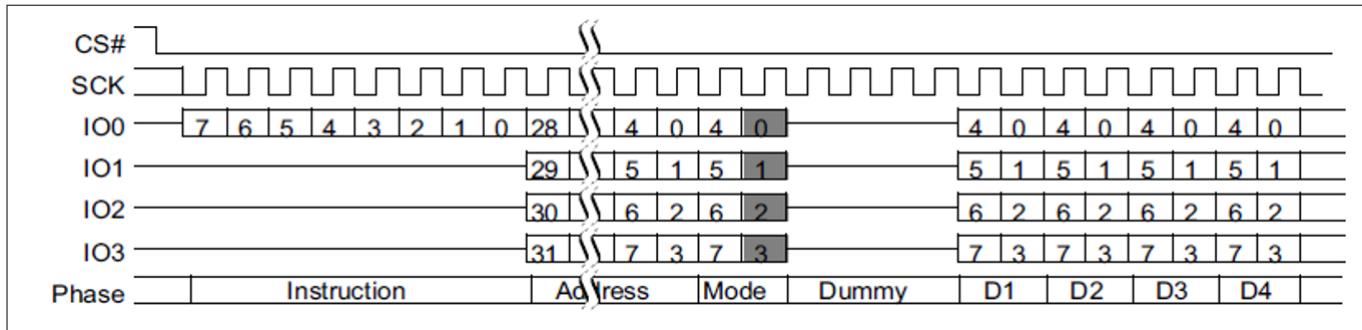


Figure 163 Quad I/O Read command sequence (4-byte address, ECh or EBh)

5.5.2.2.5 Add additional flash operation commands

To add an additional command other than the supported commands, reverse the top-down approach. This is demonstrated with the RDID (9Fh) command.

1. Check the command sequence for RDID 9Fh

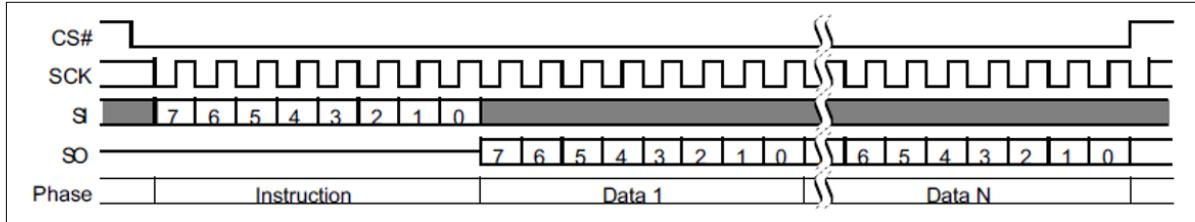


Figure 164 Read Identification (RDID 9Fh) command sequence

2. Specify the arguments for the operation command in cycfg_qspi_memslot.c to define the SMIF-driver memory configuration

5 PSoC™ 6 application notes

Code Listing 3: S25FL512S_4byteaddr_SlaveSlot_0_readIDCmd in cycfg_qspi_memslot.c

```
const cy_stc_smif_mem_cmd_t
S25FL512S_4byteaddr_SlaveSlot_0_readIDCmd =
{
    /* The 8-bit command. 1 x I/O read command. */
    .command = 0x9FU,
    /* The width of the command transfer. */
    .cmdWidth = CY_SMIF_WIDTH_SINGLE,
    /* The width of the address transfer. */
    .addrWidth = CY_SMIF_WIDTH_NA,
    /* The 8-bit mode byte. This value is 0xFFFFFFFF when there is no mode present. */
    .mode = 0xFFFFFFFFU,
    /* The width of the mode command transfer. */
    .modeWidth = CY_SMIF_WIDTH_NA,
    /* The number of dummy cycles. A zero value suggests no dummy cycles. */
    .dummyCycles = 0U,
    /* The width of the data transfer. */
    .dataWidth = CY_SMIF_WIDTH_SINGLE
};
```

3. Add the read ID command to the command structure in cycfg_qspi_memslot.c

Code Listing 4: deviceCfg_S25FL512S_4byteaddr_SlaveSlot_0 in cycfg_qspi_memslot.c

```
const cy_stc_smif_mem_device_cfg_t
deviceCfg_S25FL512S_4byteaddr_SlaveSlot_0 =
{
    ...
    .readIDCmd =
    (cy_stc_smif_mem_cmd_t*)&S25FL512S_4byteaddr_SlaveSlot_0_readIDCmd,
    ...
}
```

4. Add the function that actually operates the command in cy_smif_memslot.c where the source code for memory-level APIs for the SMIF driver is provided

5 PSoC™ 6 application notes

~~DRAFT~~ Code Listing 5: Cy_SMIF_MemCmdReadID in cy_smif_memslot.c

```

cy_en_smif_status_t Cy_SMIF_MemCmdReadID(SMIF_Type *base,
                                           cy_stc_smif_mem_config_t const *memDevice,
                                           uint8_t* readBuff,
                                           uint32_t size,
                                           cy_stc_smif_context_t *context)
{
    cy_en_smif_status_t result = CY_SMIF_BAD_PARAM;
    cy_en_smif_slave_select_t slaveSelected;
    cy_stc_smif_mem_device_cfg_t *device = memDevice->deviceCfg;
    cy_stc_smif_mem_cmd_t *cmdReadID = device->readIDCmd;
    if(NULL == cmdReadID)
    {
        result = CY_SMIF_CMD_NOT_FOUND;
    }
    else
    {
        slaveSelected = (0U == memDevice->dualQuadSlots)?memDevice->slaveSelect:
                      (cy_en_smif_slave_select_t)memDevice->dualQuadSlots;
        result = Cy_SMIF_TransmitCommand( base,
                                         (uint8_t)cmdReadID->command,
                                         cmdReadID->cmbdWidth,
                                         CY_SMIF_CMD_WITHOUT_PARAM, CY_SMIF_CMD_WITHOUT_PARAM,
                                         CY_SMIF_WIDTH_NA,
                                         slaveSelected, CY_SMIF_TX_NOT_LAST_BYTE, context);
        if(CY_SMIF_SUCCESS == result)
        {
            result = Cy_SMIF_ReceiveData(base, readBuff, size,
                                         cmdReadID->dataWidth, NULL, context);
        }
    }
    return(result);
}

```

5. Add the function that returns the data to the rxBuffer in cy_smif_memslot.c where the source code for memory-level APIs for the SMIF driver is provided

5 PSoC™ 6 application notes

Code Listing 6: Cy_SMIF_MemCmdReadID in cy_smif_memslot.c

```

DRAFT
cy_en_smif_status_t Cy_SMIF_MemReadID(SMIF_Type *base,
                                         cy_stc_smif_mem_config_t const *memConfig,
                                         uint8_t rxBuffer[],
                                         uint32_t length,
                                         cy_stc_smif_context_t *context)
{
    cy_en_smif_status_t status = CY_SMIF_BAD_PARAM;
    uint32_t chunk = 0UL;
    CY_ASSERT_L1(NULL != memConfig);
    CY_ASSERT_L1(NULL != rxBuffer);
    if(1)
    {
        /* SMIF can read only up to 65536 bytes in one go. Split the larger
         * read into multiple chunks */
        while (length > 0UL)
        {
            /* Get the number of bytes which can be read during one operation */
            chunk = (length > SMIF_MAX_RX_COUNT) ?
                    (SMIF_MAX_RX_COUNT) : length;
            /* Send the command to read data from the external memory to the rxBuffer array */
            status = Cy_SMIF_MemCmdReadID(base, memConfig,
                                          (uint8_t *)rxBuffer, chunk, context);
            if(CY_SMIF_SUCCESS == status)
            {
                /* Wait until the SMIF block completes receiving data */
                status = PollTransferStatus(base, CY_SMIF_REC_CMPLT, context);
            }
            if(CY_SMIF_SUCCESS != status)
            {
                break;
            }
            /* Recalculate the next rxBuffer offset */
            length -= chunk;
            rxBuffer = (uint8_t *)rxBuffer + chunk;
        }
    }
    return status;
}

```

6. Add the function that executes the read ID operation to the external flash in cy_serial_flash_qspi.c where it provides APIs for interacting with an external flash

5 PSoC™ 6 application notes

~~DRAFT~~ Code Listing 7: cy_serial_flash_qspi_readID in cy_serial_flash_qspi.c

```

cy_rslt_t cy_serial_flash_qspi_readID(size_t length,
                                       uint8_t* buf)
{
    cy_rslt_t result_mutex_rel = CY_RSLT_SUCCESS;
    cy_rslt_t result = _mutex_acquire();
    if (CY_RSLT_SUCCESS == result)
    {
        /* Cy_SMIF_MemReadID() returns error if (addr + length) > total flash
           size.*/
        result = (cy_rslt_t)Cy_SMIF_MemReadID(qspi_obj.base,
                                              qspi_block_config.memConfig[MEM_SLOT],
                                              buf, length,
                                              &qspi_obj.context);
        result_mutex_rel = _mutex_release();
        result_mutex_rel = _mutex_release();
    }
    /* Give priority to the status of SMIF operation when both SMIF
       operation and mutex release fail.*/
    return ((CY_RSLT_SUCCESS == result) ? result_mutex_rel :
           result);
}

```

7. In main.c, add the function to operate the read device RDID (9Fh) operation

Code Listing 8: cy_serial_flash_qspi_readID in main.c

```

int main(void)
{
...
printf("\r\n1. Reading device ID\r\n");
result = cy_serial_flash_qspi_readID(DEVICE_ID, rx_buf);
check_status("Reading Device ID failed", result);
printf("Byte Address: Data\r\n");
for(uint8_t i = 0; i < DEVICE_ID; i++)
    printf("0x%02dh : 0x%02Xh \r\n", i, rx_buf[i]);
...
}

```

Make sure to include the added functions to the header files also.

This example code is in SPI mode. Some flash devices such as S25FS512S supports QPI mode, so if you want to operate RDID (9Fh) in QPI mode, you can modify the command width and data width into Quad.

5 PSoC™ 6 application notes

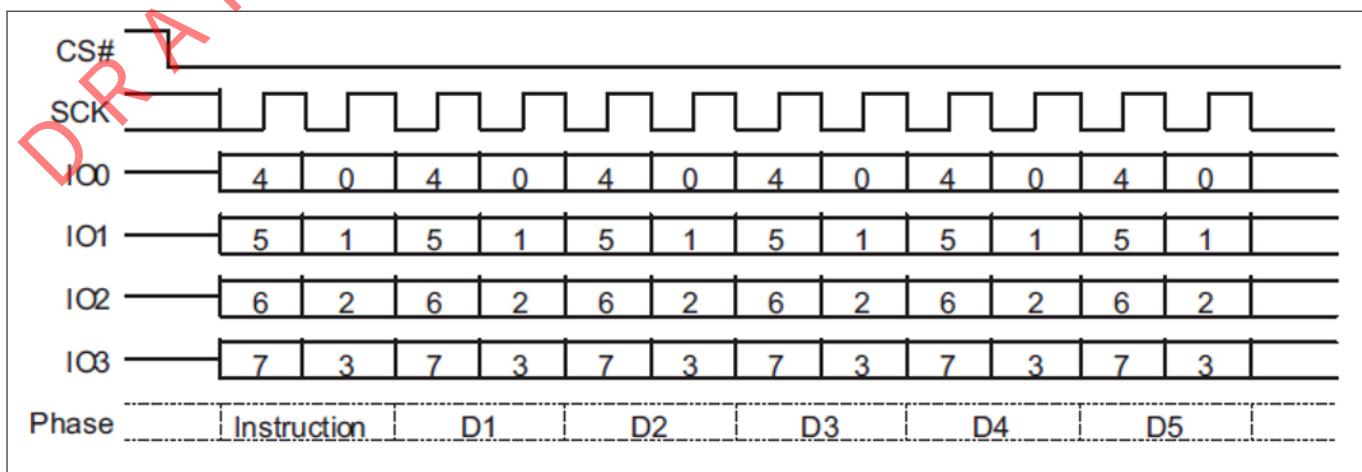


Figure 165 **Read Identification (RDID) QPI mode command**

Code Listing 9: RDID(9Fh) in QPI mode

```

const cy_stc_smif_mem_cmd_t
S25FL512S_4byteaddr_SlaveSlot_0_readIDQPICmd =
{
    /* The 8-bit command. 1 x I/O read command. */
    .command = 0x9FU,
    /* The width of the command transfer. */
    .cmdWidth = CY_SMIF_WIDTH_QUAD,
    /* The width of the address transfer. */
    .addrWidth = CY_SMIF_WIDTH_NA,
    /* The 8-bit mode byte. This value is 0xFFFFFFFF when there
       is no mode present. */
    .mode = 0xFFFFFFFFU,
    /* The width of the mode command transfer. */
    .modeWidth = CY_SMIF_WIDTH_NA,
    /* The number of dummy cycles. A zero value suggests no dummy cycles. */
    .dummyCycles = 0U,
    /* The width of the data transfer. */
};

```

~~DO NOT USE~~ 5 PSoC™ 6 application notes

5.5.3 PSoC™ 6 MCU board operation

Figure 166 shows the waveforms with the QPI bit set for the FS512S RDID QPI operation signals measured with an oscilloscope. Normally, input signal instructions are latched on the rising edge of the SCK signal. Then, the data output changes after the falling edge of SCK in SDR commands. Because this is a QPI operation, the output data should come after the first 8-bits of the two serial clock instructions on the falling edge. However, for this PSoC™ 6 MCU board, the signals remain HIGH even after the instruction phase; the data phase comes after a certain hold time. This hold time is driven by the controller.

Even though the flash memory is sending data signals after the instruction phase, the controller's pull-up signal is more dominant, thus ignoring the flash device's first data signal. Therefore, as soon as the controller drops the bus, a dip can be made regardless of the real signal due to the controller's sudden drop from HIGH to LOW. Therefore, in the oscilloscope measurements' point of view, the first data should be read right after the dip but before the second falling edge of the flash device-driven phase. This is because the flash device's output data is already on the signal bus but ignored due to controller's control over the bus.

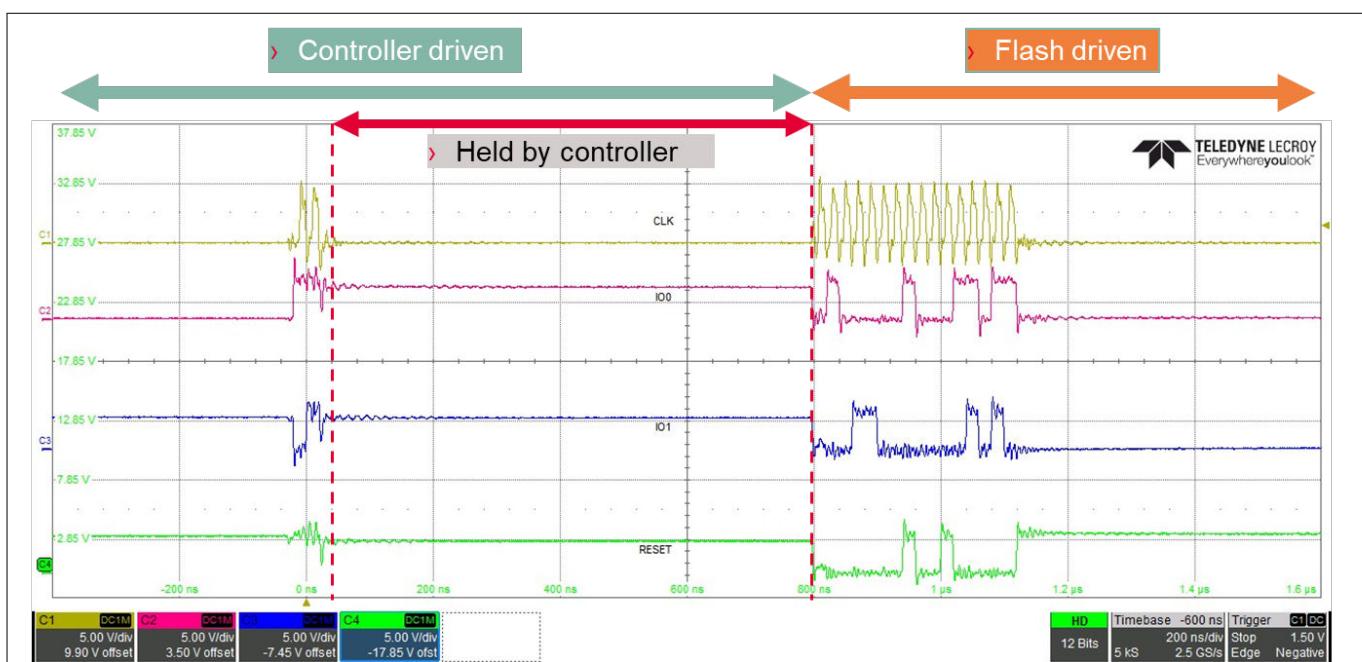


Figure 166 FS512S QPI bit set RDID(9Fh) QPI operation

5 PSoC™ 6 application notes

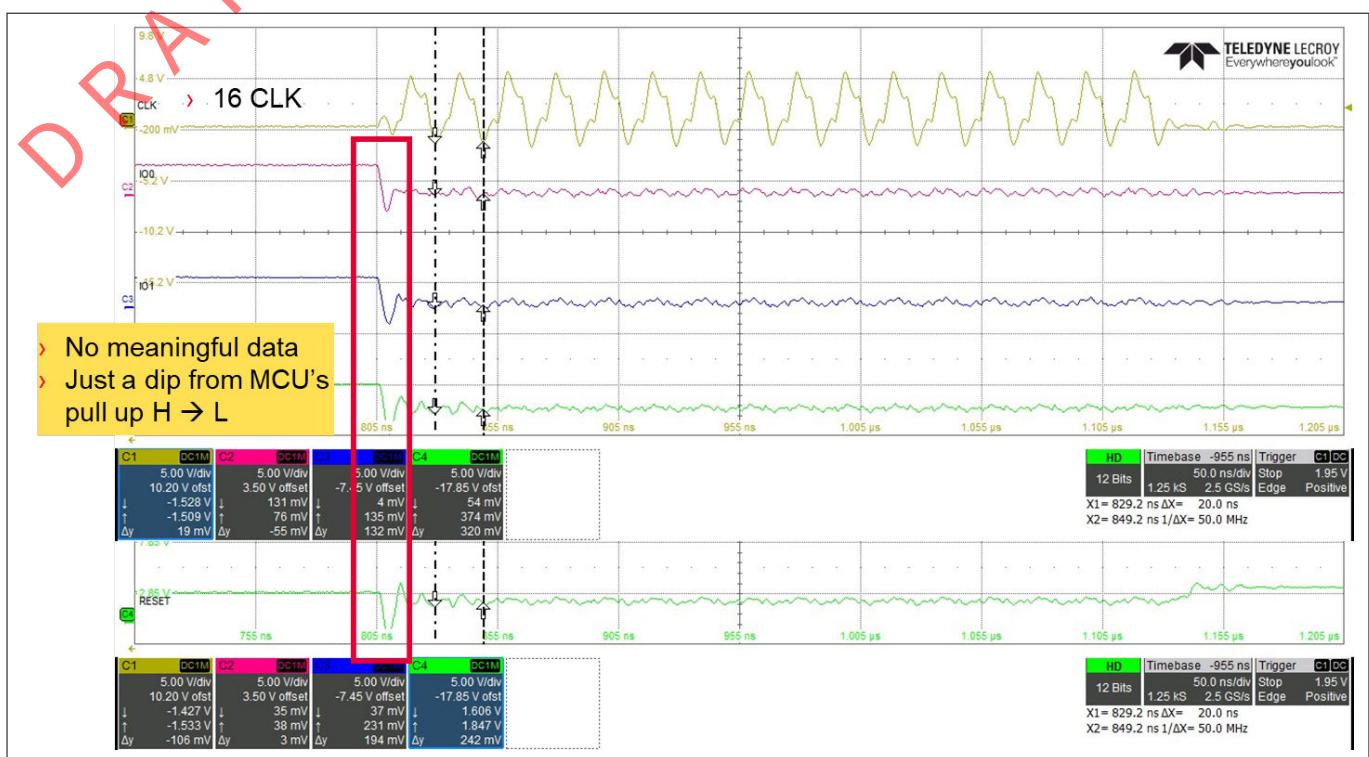


Figure 167

No meaningful dip at the beginning of the data phase

5 PSoC™ 6 application notes

5.5.4 Use cases

Even though these changes make the kit a general flash evaluation tool, its usage can be expanded beyond a demo board. Here's an actual use case to a customer's board. When a failure case is submitted, application-level investigation should be done to narrow down the root cause of the failure before failure analysis (FA) is done at the chip level.

Note that the jumpers are connected to the connector pinout on the PCB; it is then connected to the pin header of the PSoC™ 6 MCU board to avoid damage on the target board.

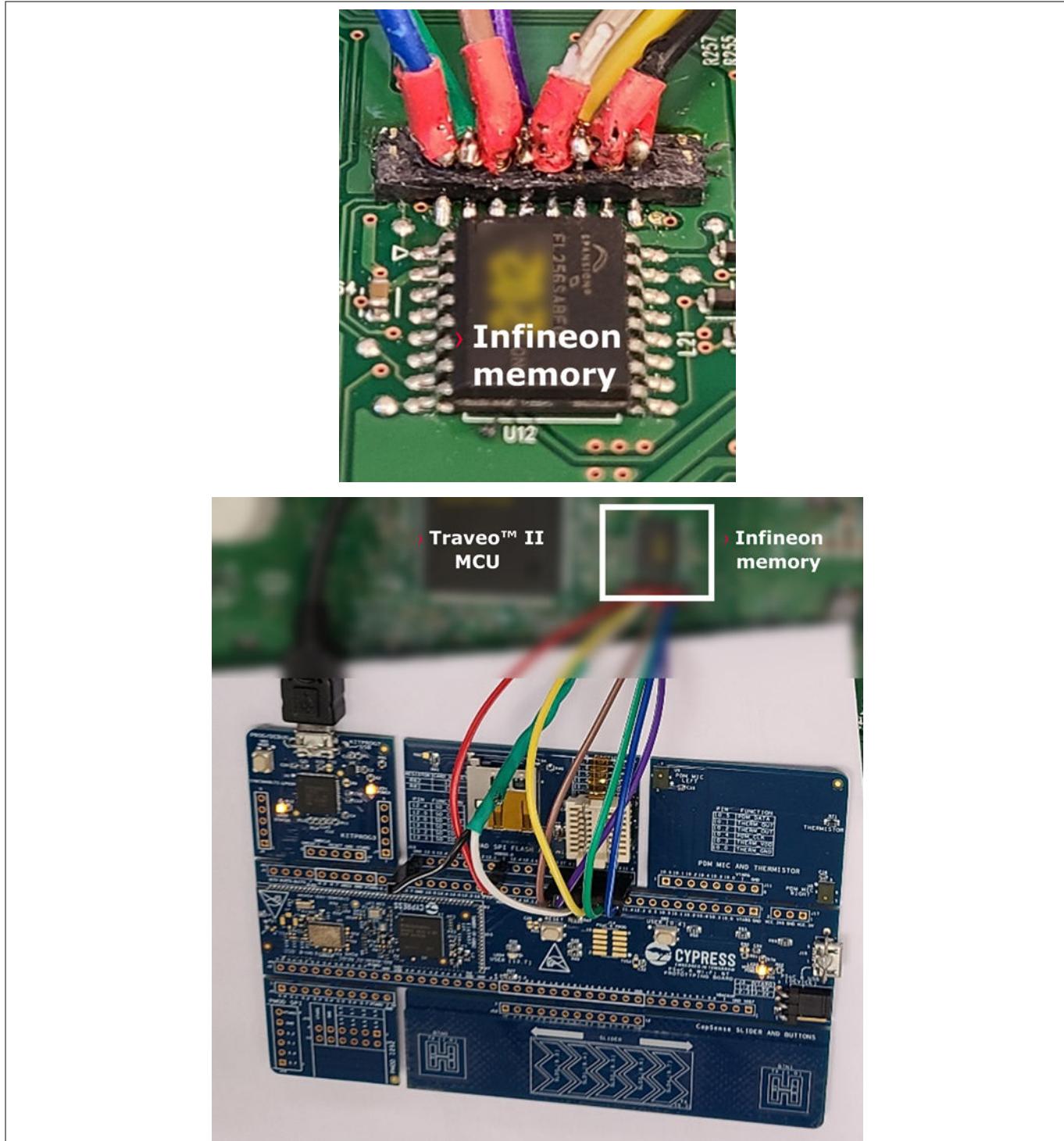


Figure 168

PSoC™ 6 MCU board connected externally to a customer's application board

5 PSoC™ 6 application notes

By reading the device ID using the function added and reading the data on a specific address, identical abnormal operation was reproduced with PSoC™ 6 MCU board.

DRAFT

```
***** PSoC 6 S&T Motiv Debugging 20210506*****
*****
***** Customer's S25FL256S *****

0. Reading ID
Reading ID
2 (6 bytes):
-----
0x01 0x02 0x19 0x40 0x00 0x80
2. Reading External memory data address: 0x0000
Received Data (64 bytes):
-----
0x54 0x41 0x43 0x53 0x31 0x38 0x30 0x31 0x30 0x32 0x30 0x30 0x34 0x30 0x30
0x30 0x30 0x30 0x30 0x35 0x32 0x30 0x30 0x39 0x30 0x32 0x30 0x38 0x33
0x01 0x00 0x02 0x00 0x04 0x00 0x05 0x00 0x06 0x00 0x07 0x00 0x08 0x00 0x09 0x00
0x0A 0x00 0x08 0x00 0x0E 0x00 0x0F 0x00 0x10 0x00 0x11 0x00 0x12 0x00 0x13 0x00
3. Reading External memory data address: 0x20000
Received Data (64 bytes):
-----
0x54 0x41 0x43 0x53 0x31 0x38 0x30 0x31 0x30 0x32 0x30 0x30 0x34 0x30 0x30
0x30 0x30 0x30 0x30 0x35 0x32 0x30 0x30 0x39 0x30 0x32 0x30 0x38 0x33
0x01 0x00 0x02 0x00 0x04 0x00 0x05 0x00 0x06 0x00 0x07 0x00 0x08 0x00 0x09 0x00
0x0A 0x00 0x08 0x00 0x0E 0x00 0x0F 0x00 0x10 0x00 0x11 0x00 0x12 0x00 0x13 0x00
4. Reading External memory data address: 0x40000
Received Data (64 bytes):
-----
0x54 0x41 0x43 0x53 0x31 0x38 0x30 0x31 0x30 0x32 0x30 0x30 0x34 0x30 0x30
0x30 0x30 0x30 0x30 0x35 0x32 0x30 0x30 0x39 0x30 0x32 0x30 0x38 0x33
0x01 0x00 0x02 0x00 0x04 0x00 0x05 0x00 0x06 0x00 0x07 0x00 0x08 0x00 0x09 0x00
0x0A 0x00 0x08 0x00 0x0E 0x00 0x0F 0x00 0x10 0x00 0x11 0x00 0x12 0x00 0x13 0x00
5. Reading External memory data address: 0x60000
Received Data (64 bytes):
-----
0x54 0x41 0x43 0x53 0x31 0x38 0x30 0x31 0x30 0x32 0x30 0x30 0x34 0x30 0x30
0x30 0x30 0x30 0x30 0x35 0x32 0x30 0x30 0x39 0x30 0x32 0x30 0x38 0x33
0x01 0x00 0x02 0x00 0x04 0x00 0x05 0x00 0x06 0x00 0x07 0x00 0x08 0x00 0x09 0x00
0x0A 0x00 0x08 0x00 0x0E 0x00 0x0F 0x00 0x10 0x00 0x11 0x00 0x12 0x00 0x13 0x00
[]
```

Figure 169 Read ID operation result

5 PSoC™ 6 application notes~~DRAFT~~
5.5.5 Conclusion

Even though CY8CPROTO-062-4343W PSoC™ 6 prototyping kit is provided with a Quad SPI flash device and example code, its use as a general-purpose flash evaluation tool is limited. By modifying the hardware and software of the board, it can be turned into a general tool to operate any flash device.

By attaching a socket, the flash device can be replaced with any SOIC package flash. By attaching pin headers, it frees the spatial constraints and connects any flash device with any package and even to the flash device on an external board. By adding operation commands, the test range can be extended to any commands.

By using the information in this application note, you will be able to make your own PSoC™ 6 MCU-based flash evaluation tool.

5 PSoC™ 6 application notes

References

-
- 5
- 1. PSoC™ 6 Wi-Fi Bluetooth® prototyping kit (CY8CPROTO-062-4343W)
<https://www.cypress.com/documentation/development-kitsboards/psoc-6-wi-fi-bt-prototyping-kit-cy8cproto-062-4343w>
- 2. CY8CPROTO-062-4343W PSoC™ 6 Wi-Fi Bluetooth® prototyping kit guide
<https://www.cypress.com/file/457891/download>
- 3. CY8CPROTO-062-4343W schematic
<https://www.cypress.com/file/457811/download>
- 4. S25FL512S, 512 Mb (64 MB), 3.0 V SPI flash memory datasheet
<https://www.cypress.com/documentation/datasheets/s25fl512s-512-mb-64-mb-30-v-spi-flash-memory>
- 5. ModusToolbox™ software download link
https://www.cypress.com/products/modustoolbox#tabs-0-bottom_side-6
- 6. ModusToolbox™ software installation guide
<https://www.cypress.com/file/520241/download>
- 7. ModusToolbox™ software user guide
<https://www.cypress.com/file/520251/download>

5 PSoC™ 6 application notes**5.5.6 Revision history**

Document version	Date of release	Description of changes
**	2021-09-17	New application note
*A	2022-07-21	Template update

5.6 AN228740 Usage of Quad SPI (QSPI)/Serial Memory Interface (SMIF) in PSoC™ 6 MCU**About this document**

-
- 6

Scope and purpose

AN228740 provides the guidelines for using QSPI/SMIF in PSoC™ 6 MCU. The PSoC™ 6 MCU QSPI delivers an interface for communicating with external serial memory devices. This application note explains how to incorporate QSPI into an application in either eXecute In Place (XIP) mode or Command mode. This application note describes the features available in the block and how to configure those features for your application.

Associated part family

PSoC™ 6 MCU

More code examples? We heard you.

To access an ever-growing list of hundreds of PSoC™ code examples, please visit our [code examples web page](#). You can also explore the video training library [here](#).

~~DO NOT USE~~

5 PSoC™ 6 application notes

5.6.1 ~~DO NOT USE~~ Introduction

An MCU is used to process data from sensors or other external devices in many embedded systems. The MCU often has limited on-chip memory for data storage beyond the processing firmware that the MCU is executing. This can make processing more significant data types, such as images and audio files, complex as the file cannot fit in the MCU's memory. Additionally, as algorithms become more complex and firmware images grow in size, an MCU's on-chip memory may not be large enough. To remedy this lack of memory, external serial memory devices can be added to an embedded system to greatly increase available storage.

The PSoC™ 6 MCU includes a serial memory interface (SMIF) hardware block that simplifies access to external serial memory devices. This block supports a variety of SPI-based serial interfaces, including standard SPI, Dual-SPI, Quad-SPI, Dual Quad-SPI, and Octal SPI. The block also supports eXecute-In-Place (XIP) mode operation, so that large firmware images can be executed directly from the external memory with minimal latency.

This application note shows you how to use QSPI to communicate with serial memory devices with the Infineon PSoC™ 6 MCU device. The document opens with two simple software approaches to using QSPI. These include example code snippets that can be used as a reference to quickly get started using QSPI. More in-depth detail about QSPI and its capabilities is presented after the software examples. The topics discussed include:

- Features of QSPI
- Configuring PSoC™ 6 MCU QSPI to work with your external memory device
- On-the-Fly (OTF) encryption and decryption with QSPI
- QSPI caching
- QSPI eXecute-In-Place (XIP) and Command modes

In addition, this application note explains several system-level topics such as securing the external memory, programming external memory using programming tools, and creating configurations for use with the QSPI Configuration tool.

As most of the serial memory devices in the market support the Quad-SPI (QSPI) interface, the rest of this application note will use QSPI as a general term to refer to the memory interface and all its configurations.

This application note assumes that you are familiar with the basic PSoC™ 6 MCU architecture found in the device [datasheet](#) or the [technical reference manual \(TRM\)](#).

5 PSoC™ 6 application notes

5.6.2 Getting started with QSPI

The QSPI block provides dedicated hardware for accessing serial memory devices in SPI, Dual-SPI, Quad-SPI, Dual-Quad-SPI, and Octal-SPI modes. The block is fully supported in the [ModusToolbox™ Software Environment](#), with multiple API layers for accessing the QSPI hardware.

5.6.2.1 Using the Serial Flash Library

The [Serial Flash Library](#) is an easy way to get started with the QSPI block. This library supports all features necessary for accessing most serial flash memory devices. This library also provides simple function calls that handle most of the configuration steps automatically. As a result, the Serial Flash Library is simple to use and is suitable for most use cases. The Serial Flash Library functions call a combination of functions from [Hardware Abstraction Layer \(HAL\)](#) and [Peripheral Driver Library \(PDL\)](#).

Follow these steps to get started with the Serial Flash Library using the Eclipse IDE for ModusToolbox™.

1. Enable the library in the **Library Manager** (see [Figure 170](#))

- a. Select the Library Manager
- b. Go to the Libraries tab
- c. Select the **Serial Flash Library**
- d. Select Apply

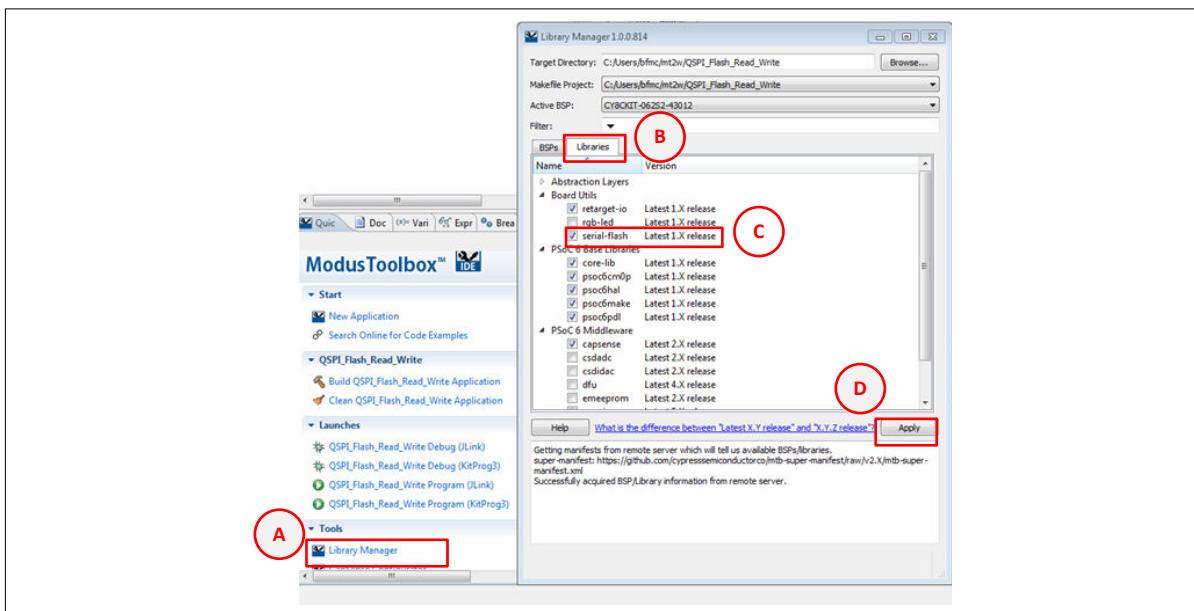


Figure 170 Adding Serial Flash Library

2. Include the necessary libraries in your application's `main.c` file

```
#include "cyhal.h" /* HAL */
#include "cycfg.h" /* Auto-generated system configuration headers */
#include "cybsp.h" /* Board specific pin and peripheral definitions */
#include "cycfg_qspi_memslot.h" /*QSPI external memory configuration structures */
#include "cy_serial_flash_qspi.h" /* QSPI library functions */
```

5 PSoC™ 6 application notes

- ~~DRAFT~~
3. In the main function, initialize the board peripherals, the UART, and the QSPI block. The QSPI configuration is included in the board support package (BSP) for your kit

```

/* Initialize the device and board peripherals */
result = cybsp_init();
CY_ASSERT(result == CY_RSLT_SUCCESS);
/* Enable global interrupts */
__enable_irq();
/* Initialize retarget-io to use the debug UART port */
cy_retarget_io_init(CYBSP_DEBUG_UART_TX, CYBSP_DEBUG_UART_RX, CY_RETARGET_IO_BAUDRATE);
/* Initialize qspi block and external memory device */
cy_serial_flash_qspi_init(smifMemConfigs[MEM_SLOT_NUM], CYBSP_QSPI_D0, CYBSP_QSPI_D1,
CYBSP_QSPI_D2, CYBSP_QSPI_D3, NC, NC, NC, NC, CYBSP_QSPI_SCK, CYBSP_QSPI_SS,
QSPI_BUS_FREQUENCY_HZ);

```

4. Create data buffers to store read and write data. Transfer the data to the external memory. Read back the data and print it to a UART terminal for verification. In the below example, the data is written to and read from the address 0x00040000 in external memory. This address is offset from the base address of external memory, not from the PSoC™ 6 MCU's local addressing range.

```

/* Read/write buffers */
uint8 rxBuffer[PACKET_SIZE];
uint8 txBuffer[PACKET_SIZE];
/* Write the content of the txBuffer to the memory */
cy_serial_flash_qspi_write(0x00040000, PACKET_SIZE, txBuffer);

/* Read back after Write for verification */
cy_serial_flash_qspi_read(0x00040000, PACKET_SIZE, rxBuffer);

```

5.6.2.2 Using the Peripheral Driver Library

PDL provides a set of functions and structures that access the registers of the QSPI block directly, enabling you to fully configure transactions performed through the QSPI block. This configurability makes the PDL suitable for use in applications where finer control is required.

To get started with the PDL for QSPI using the ModusToolbox™ IDE, do the following:

1. Enable the QSPI hardware with using the device configurator. [Figure 171](#) shows an example configuration suitable for use with any PSoC™ 6 MCU kit that has an external flash device.
 - a. Select the **Device Configurator** from the **Quick Panel**
 - b. Check the box **Quad Serial Memory Interface (QSPI)** 0 to enable it
 - c. Configure the **QSPI** block as shown in [Figure 171](#)
 - d. Select the “Go to Signal” button next to **Interface Clock** to open the clock configuration tab
 - e. Change the clock **Divider** to **2**
 - f. Save your changes and close the device configurator

5 PSoC™ 6 application notes

DRAFT

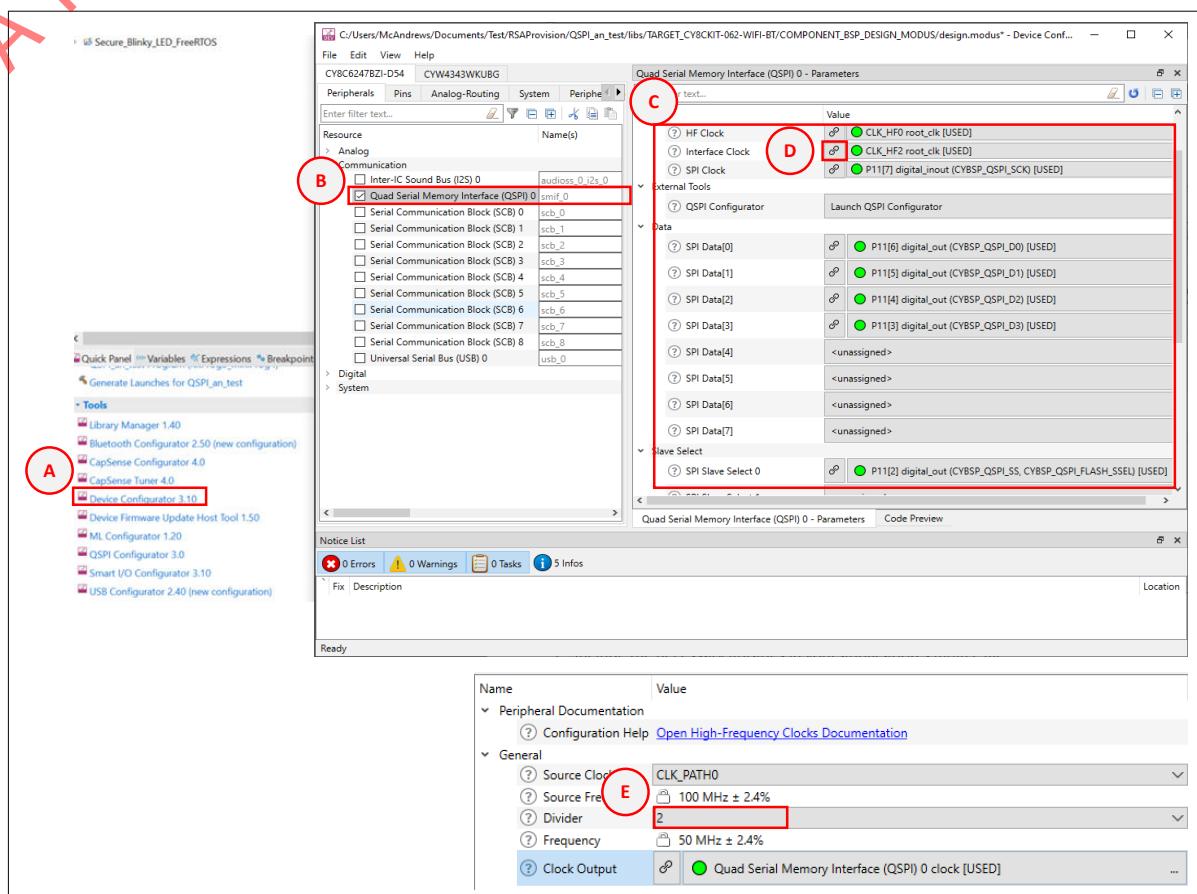


Figure 171 Steps to configure the QSPI block

2. Generate the QSPI configuration structures using the QSPI Configurator. [Figure 172](#) shows an example configuration suitable for use with any PSoC™ 6 MCU kit that has an external flash device.

Note: These instructions describe the process for using an SDFP compliant device. For more information on how to use the QSPI Configurator, including information about how you can generate code for your specific memory configuration, see the [QSPI configurator guide](#).

- a. Select the **QSPI Configurator**
- b. Select **Auto detect SDFP** from the drop-down menu
- c. Save the configuration

5 PSoC™ 6 application notes

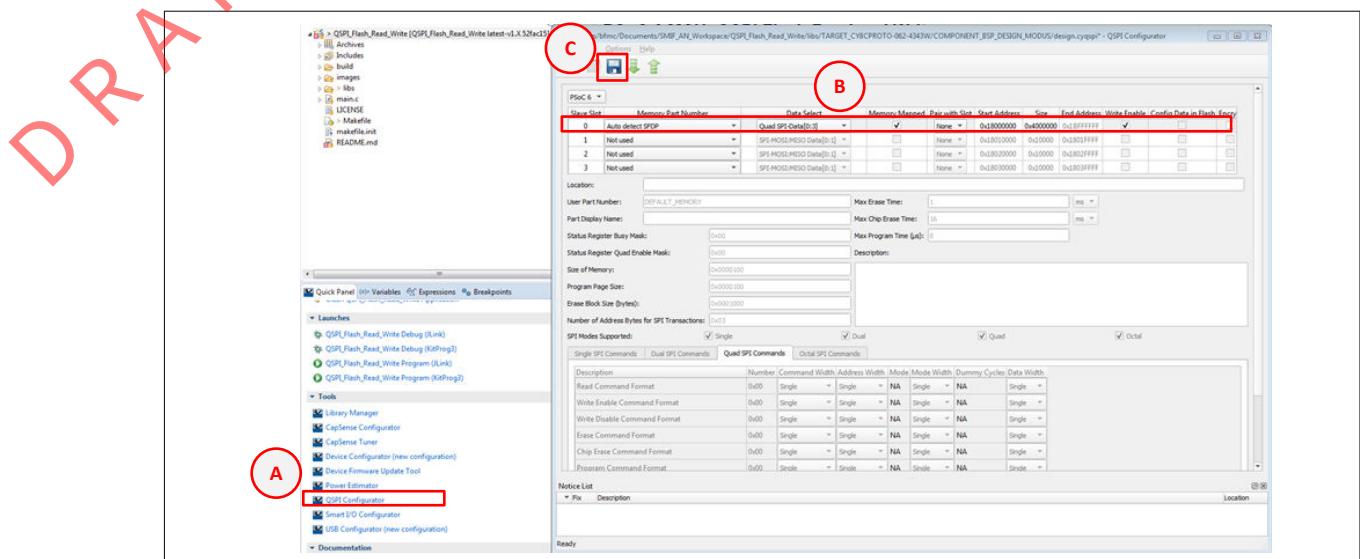


Figure 172 Steps to use the QSPI Configurator

- Include the necessary libraries in your application's main.c file.

```
#include "cy_pdl.h" /* Peripheral Driver Library */
#include "cybsp.h" /* Board specific pin and peripheral definitions */
#include "cycfg_qspi_memslot.h" /*QSPI external memory configuration structures */
```

- Create a global context variable for the QSPI block.

```
cy_stc_smif_context_t smif_context;
```

- Define a transaction packet size

```
#define PACKET_SIZE (64u)
```

- Create an interrupt routine containing a call to the QSPI API interrupt function Cy_SMIF_Interrupt. All FIFO operations will use this interrupt.

```
void SMIF_Interrupt_User(void)
{
    Cy_SMIF_Interrupt(SMIF0, &smif_context);
}
```

- In the main function, initialize the peripherals and global interrupts.

```
/* Initialize the device and board peripherals */
result = cybsp_init();
CY_ASSERT(result == CY_RSLT_SUCCESS);
/* Enable global interrupts */
__enable_irq();
```

5 PSoC™ 6 application notes

8. In the `main` function, set up the SMIF interrupts.

```
cy_stc_sysint_t smifIntConfig =  
{  
    #if (CY_CPU_CORTEX_M0P)  
        /* .intrSrc */ NvicMux7_IRQn,  
        /* .cm0pSrc */ smif_interrupt_IRQn,  
    #else  
        /* .intrSrc */ smif_interrupt_IRQn, /* SMIF interrupt number */  
    #endif  
    /* .intrPriority */ 7u  
};  
(void) Cy_SysInt_Init(&smifIntConfig, SMIF_Interrupt_User);
```

9. In the `main` function, initialize and enable the QSPI block.

```
/* SMIF initialization */  
Cy_SMIF_Init(SMIF0, &smif_0_config, TIMEOUT_1_S, &smif_context);  
Cy_SMIF_Enable(SMIF0, &smif_context); /* Enable the SMIF Interrupt */  
#if (__CORTEX_M == 0)  
    NVIC_EnableIRQ(NvicMux7_IRQn);  
#else  
    NVIC_EnableIRQ(smif_interrupt_IRQn);  
#endif
```

10. If your external memory supports Serial Flash Discoverable Parameters (SFDP), detect the parameters. For manual memory configuration, this step is optional and only required if you intend to use the device in XIP mode.

```
/* Memslot level initialization */  
Cy_SMIF_MemInit(SMIF0, &smifBlockConfig, &smif_context);
```

5 PSoC™ 6 application notes

- DRAFT**
11. If your external memory device supports Quad mode, enable Quad mode.

```
bool isQuadEnabled = false;
Cy_SMIF_MemIsQuadEnabled(SMIF0, smifBlockConfig.memConfig[0],
                           &isQuadEnabled,
                           &smif_context);

Cy_SMIF_MemEnableQuadMode(SMIF0,
                           smifBlockConfig.memConfig[0],
                           5000,
                           &smif_context);
```

12. Begin transacting with the external memory device. The following is an example of a write sequence sending data from txBuffer.

```
uint8_t txBuffer[PACKET_SIZE];
uint8_t rxBuffer[PACKET_SIZE];
/* Erase before write */
Cy_SMIF_MemEraseSector(SMIF0, smifMemConfigs[0],
                       0x00,
                       smifMemConfigs[0]->deviceCfg->eraseSize,
                       &smif_context);

/* Read to rxBuffer after Erase to confirm that all data is 0xFF */
Cy_SMIF_MemRead(SMIF0, smifMemConfigs[0],
                  smifMemConfigs[0]->deviceCfg->eraseSize,
                  rxBuffer,
                  PACKET_SIZE,
                  &smif_context);

/* Write txBuffer to the external memory */
Cy_SMIF_MemWrite(SMIF0, smifMemConfigs[0],
                  smifMemConfigs[0]->deviceCfg->eraseSize,
                  txBuffer,
                  PACKET_SIZE,
                  &smif_context);
```

5 PSoC™ 6 application notes

5.6.3 Features of QSPI

~~DRAFT~~
QSPI provides a highly configurable interface between PSoC™ 6 MCU and an external serial memory device. The QSPI block contains several sub-components which enable caching, XIP mode, Command mode, and cryptography.

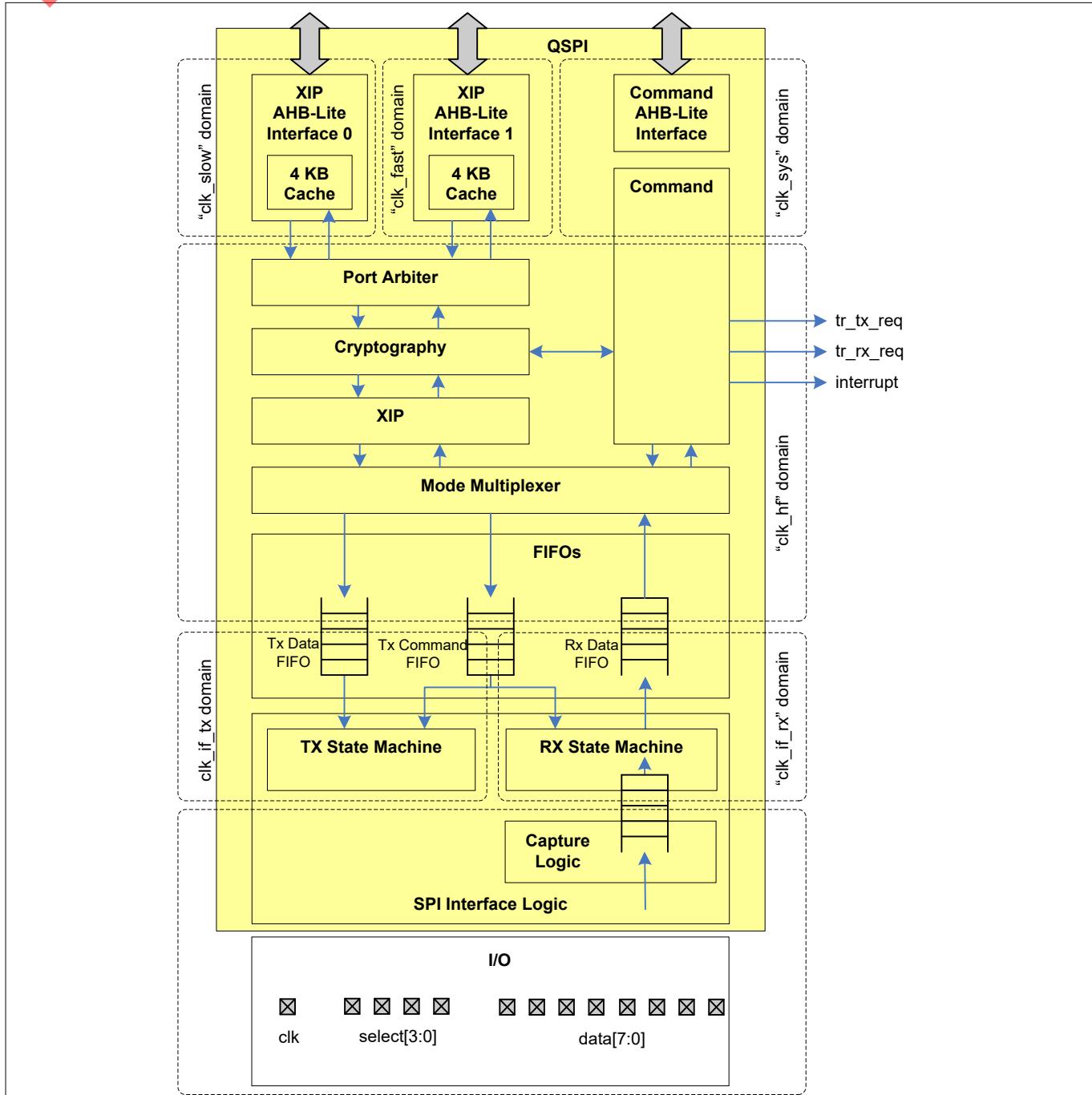


Figure 173 QSPI block architecture

5.6.3.1 Clock domains

The QSPI block has three AHB-Lite bus interfaces: two for XIP mode and one for Command mode. The XIP mode interfaces consist of a fast domain and a slow domain. Arm® Cortex®-M4 is the only bus master in the fast domain. In the slow domain, Cortex®-M0, Crypto, Datawire0, and Datawire1 can be bus masters. For Command

5 PSoC™ 6 application notes

~~DRAFT~~
mode operation, the bus interface is in the `c1k_sys` domain, which is a divided clock from `c1k_hf`. In Command mode, the bus master can be any of the bus masters in XIP mode.

The FIFOs in the block work in the SPI interface clock domain. The remainder of the block components, including the cryptography, mode multiplexer, and port arbiter operate in the high frequency clock domain.

5.6.3.2 Modes

The QSPI hardware provides a mode multiplexer, which allows you to operate the QSPI block in either Command mode or XIP mode. In the PDL, this mode can be changed during runtime by calling the `cy_QSPI_SetMode()` function. Note that the mode should be changed only after any ongoing transfers are completed. A call to the `cy_SMIF_BusyCheck()` function should be made to ensure that the QSPI block is not busy.

5.6.3.2.1 Command mode

This is the default mode of the QSPI block and is typically used for large data storage. In this mode, data transfers are initiated by accessing the FIFOs. Software may transfer command bytes to the TX command FIFO and data bytes to the TX and RX FIFOs. This mode generates triggers depending on the number of FIFO entries available or used. The trigger `tr_tx_req` is active when the TX data FIFO has fewer entries than specified by the `TX_DATA_FIFO_CTL`. `TRIGGER_LEVEL` field. The trigger `tr_rx_req` is active when the RX data FIFO has more entries than specified by the `RX_DATA_FIFO_CTL`. `TRIGGER_LEVEL` field.

Command mode provides the flexibility to implement any SPI transfer, including transfers to configure or erase the external memory. However, in Command mode, your application code must generate the opcode, slave address, dummy cycles, and data. This results in many CPU cycles being spent for each transaction. These extra cycles are not desirable for accessing small amounts of data, such as executing code. However, in this mode, a single transaction can transfer up to 65535 bytes. Hence, Command mode is recommended for bulk data transfers or infrequently accessed data, such as images or other large data types.

5.6.3.2.2 XIP mode

This mode is typically used to execute code out of an external memory device. In this mode, the QSPI block automatically generates SPI transfers without software intervention. The external memory space is mapped to a configurable range of addresses in the PSoC™ 6 MCU's address space through one of the two XIP AHB-lite interfaces. As a result, external memory accesses in XIP mode do not require discrete software intervention and data stored in external memory can be accessed like any other variable.

5.6.3 Caches

The QSPI block also has a dedicated 4 KB cache for each of the XIP interfaces; one for CM4 and another for CM0 and DMA. These caches are enabled by default. Read transfers that hit in the cache are processed by the cache, while read transfers that miss in the cache incur an XIP memory read transfer of 16 bytes to refill the missed subsector. There is also a prefetch buffer, which grabs the next 16 bytes of data to refill the cache. This means that XIP transfers will occur in 32-byte chunks, 16 bytes for the cache and 16 bytes from the prefetch buffer, before a SPI transfer occurs to refill the cache subsector and the prefetch buffer.

As a result, XIP mode is convenient and efficient for small transfers such as executing code out of external memory. For large reads, however, cache and prefetch buffer refills require repeat transmission of the opcode, device address, and dummy cycles every 32 bytes. These refills add latency to the transfer that would not exist in the Command mode.

For RAM devices, XIP mode can write to the external memory. Write transfers in XIP mode bypass the cache and incur a SPI transfer. If a bus master reads and writes to the external memory device, the writes will automatically invalidate the cache.

~~5 PSoC™ 6 application notes~~

It is possible for data in the caches to be invalid when data is written to a location in the XIP addressing range. To avoid reading stale data from the caches, you should invalidate the cache when new data is written to the XIP memory region. An example of this would be if you were executing code out of external memory at address 0x18000000 when the QSPI block was transitioned into Command mode and a write occurred at the same address. Switching the block back into XIP mode and executing from that address could cause a failure. It is also possible for two masters from different AHB clock domains to access the external memory through the SMIF block, such as the DMA hardware and CM4. In this use case, it is a good practice to make sure the regions being accessed by each master do not overlap. This minimizes the likelihood that the cache will contain invalid data. If the accessed regions do overlap, the cache can be disabled with the PDL function call `Cy_QSPI_CacheDisable()` or the cache will need to be invalidated after each write using `Cy_QSPI_CacheInvalidate()`.

5.6.3.4 Memory device signal interface

The QSPI block acts as a SPI master when communicating with external memory devices. QSPI only supports SPI configuration 0, where the clock polarity (CPOL) is 0 and the clock phase (CPHA) is 0. In addition, to standard SPI, the block is also capable of operating in Dual-SPI, Quad-SPI, Dual Quad-SPI, and Octal-SPI modes. For all modes, the block operates using Single Data Rate (SDR) mode.

QSPI can support up to four memory devices simultaneously, limited by the number of data select lines. For example, QSPI supports eight data lines, which means that four single or Dual-SPI memory devices can use all eight data lines and all of the available data select lines, or the same four memory devices can use the same data lines but different data select lines. Likewise, four Quad-SPI or Octal-SPI devices can be used simultaneously, sharing data lines but using unique data selects. For a given memory device, the data lines used must be adjacent. To see the signals used by specific memory device types, see [Table 19](#).

Table 19 SPI clock, select, and data lines used for different memory devices

Memory device	I/O signals
Single SPI memory	SCK, CS, SI, SO. This memory device has two data signals (SI and SO).
Dual SPI memory	SCK, CS, IO0, IO1. This memory device has two data signals (IO0 and IO1).
Quad SPI memory	SCK, CS, IO0, IO1, IO2, IO3. This memory device has four data signals (IO0, IO1, IO2, IO3).
Octal SPI memory	SCK, CS, IO0, IO1, IO2, IO3, IO4, IO5, IO6, IO7. This memory device has eight data signals (IO0, IO1, IO2, IO3, IO4, IO5, IO6, IO7).

Each memory device must be mapped to one of the four “slots” in the QSPI block. The slot for your memory device will have a corresponding I/O pin controlling the CS line. To ensure that firmware accesses the correct device, make sure that your QSPI configuration uses the same CS line to which your external memory CS is connected.

For each of the four select lines, there are legal and illegal configurations. [Table 20](#) lists the legal configurations and the corresponding data select enumerated type defined in `cy_smif.h`. This data set is used automatically when configuring your device using the QSPI Configurator.

Table 20 Data select lines and available device configurations

cy_en_smif_data_select_t	Single SPI device	Dual SPI device	Quad SPI device	Octal SPI device
CY_SMIF_DATA_SEL0	spi_data[0] = SI spi_data[1] = SO	spi_data[0] = IO0 spi_data[1] = IO1	spi_data[0] = IO0 ... spi_data[3] = IO3	spi_data[0] = IO0 ... spi_data[7] = IO7
CY_SMIF_DATA_SEL1	spi_data[2] = SI spi_data[3] = SO	spi_data[2] = IO0 spi_data[3] = IO1	Illegal	Illegal

(table continues...)

5 PSoC™ 6 application notes

~~DRAFT~~ **Table 20** (continued) Data select lines and available device configurations

cy_en_smif_data_select_t	Single SPI device	Dual SPI device	Quad SPI device	Octal SPI device
CY_SMIF_DATA_SEL2	spi_data[4] = SI spi_data[5] = SO	spi_data[4] = IO0 spi_data[5] = IO1 ... spi_data[7] = IO3	spi_data[4] = IO0 ... spi_data[7] = IO3	Illegal
CY_SMIF_DATA_SEL3	spi_data[6] = SI spi_data[7] = SO	spi_data[6] = IO0 spi_data[7] = IO1	Illegal	Illegal

The spi_data values correspond to the QSPI I/O pins on your PSoC™ 6 MCU device. To determine which pins are available as spi_data pins for your device, see the Alternate Pin Function section of the device [datasheet](#).

Dual-quad configurations are also supported by QSPI. In dual-quad configuration, two quad-SPI devices are used simultaneously, with each device contributing a nibble of a byte per transfer. The devices will share the interface clock signal, but will use different CS lines and separate I/O lines. [Table 21](#) lists the configuration for dual-quad mode.

Table 21 Dual-quad SPI configuration

DATA_SEL[1:0]		Dual-Quad SPI Configuration	
CY_SMIF_DATA_SEL0	CY_SMIF_DATA_SEL2	spi_data[0] = IO0 ... spi_data[3] = IO3	spi_data[4] = IO4 ... spi_data[7] = IO7

For more information about the legal configurations and example diagrams of proper configurations, see the PSoC™ 6 MCU [architecture TRM](#).

5.6.3.5 Cryptography

The QSPI block includes a cryptography component to make sure data can be securely stored in external memory. The encryption and decryption are based on the AES-128 forward block cipher. A 128-bit key, stored in dedicated write-only QSPI registers SMIF_CRYPTO_KEY3, ..., SMIF_CRYPTO_KEY0, is used with a 128-bit plaintext to generate a ciphertext. The method of generating the ciphertext depends on whether the QSPI block is in Command or XIP mode.

The Serial Flash Library does not support encryption, so the PDL functions for encryption must be used. If you prefer the Serial Flash Library for configuration and data transfers, a combination of Serial Flash Library and PDL can be used.

5.6.3.5.1 Cryptography in XIP mode

In XIP mode, the cryptography component supports on-the-fly encryption and decryption, which is applied automatically on the code executed from an external memory or data written to or read from an external RAM. The encryption uses AES-128 encryption algorithm with your key on input data called a plaintext. The plaintext in XIP mode is the 28-bit XIP address extended to 128 bits with the contents of the SMIF0_CRYPTO_INPUT registers.

5 PSoC™ 6 application notes

DRAFT

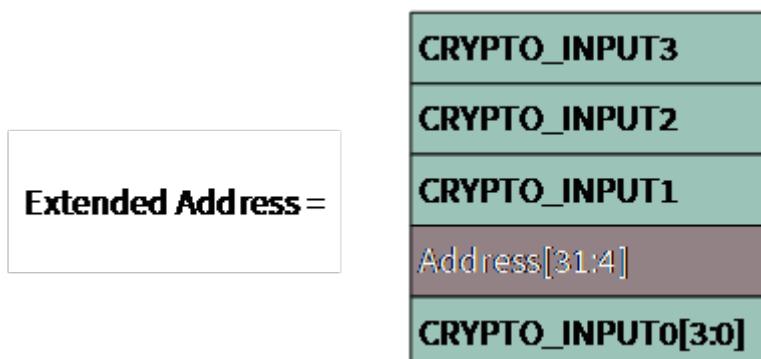


Figure 174 shows the format of the extended address.

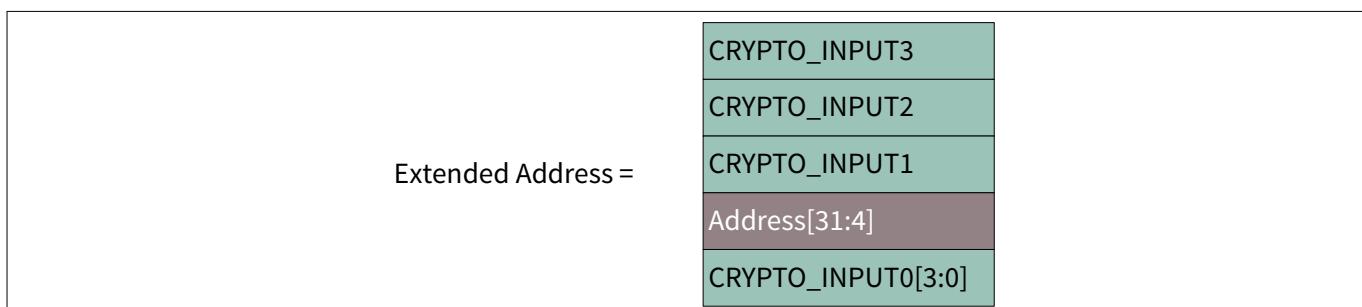


Figure 174 Extended address used in XIP cryptography

After applying AES-128 with your key on the extended address, the resulting ciphertext is XOR'd with the transfer's read or write data. By applying AES-128 to the address rather than the data being transferred, the encryption and decryption occur on-the-fly and cause no delay. Figure 175 shows the entire encryption process in XIP mode.

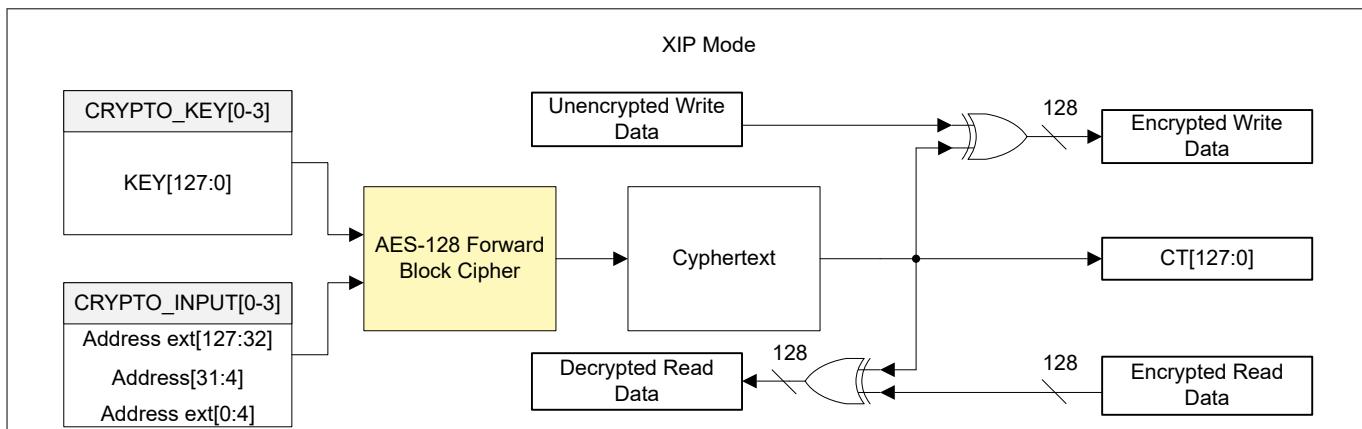


Figure 175 On-the-fly encryption process for XIP mode

To enable encryption in XIP mode, set the SMIF0_DEVICEEn_CTL.CRYPTO_EN bit, where DEVICEEn refers to the memory slot of your external memory device. For example, to enable encryption on a memory device in slot 2, use the following line of code:

```
SMIF0->DEVICE[2].CTL |= (1 << SMIF_DEVICE_CTL_CRYPTO_EN_Pos /* 8U */);
```

To disable encryption, clear the CRYPTO_EN bit.

5 PSoC™ 6 application notes~~DRAFT~~
5.6.3.5.2 Cryptography in Command mode

Command mode cryptography can be used to encrypt bootloaders, application images, or bulk data in external flash memories. Encryption in Command mode requires discrete calls to the PDL encryption function, Cy_SMIF_Encrypt, for each data transfer. The resulting ciphertext is stored in the CRYPTO_OUTPUT registers and must be unpacked before writing the data to external memory, but this is handled inside the Cy_SMIF_Encrypt function automatically. For compatibility with XIP on-the-fly decryption, the Cy_SMIF_Encrypt function uses the same encryption scheme used in XIP mode encryption.

For Command mode, the encryption flow should generally follow these steps:

1. Make sure you have loaded your encryption key into the SMIF_CRYPTO_KEY registers.
2. Encrypt the data to be transferred using the PDL function Cy_SMIF_Encrypt.
3. Write the data to the external memory.

For decryption:

1. Read the encrypted data from the external memory.
2. Decrypt the data using the PDL function Cy_SMIF_Encrypt.

~~5 PSoC™ 6 application notes~~

~~5.6.4 Ecosystem~~

The QSPI block has an ecosystem of tools and files consisting of:

- ModusToolbox™ Application Software libraries
- The QSPI Configurator tool
- Programming tools

Figure 176 shows the various components of the ecosystem and the files used or generated by those components.

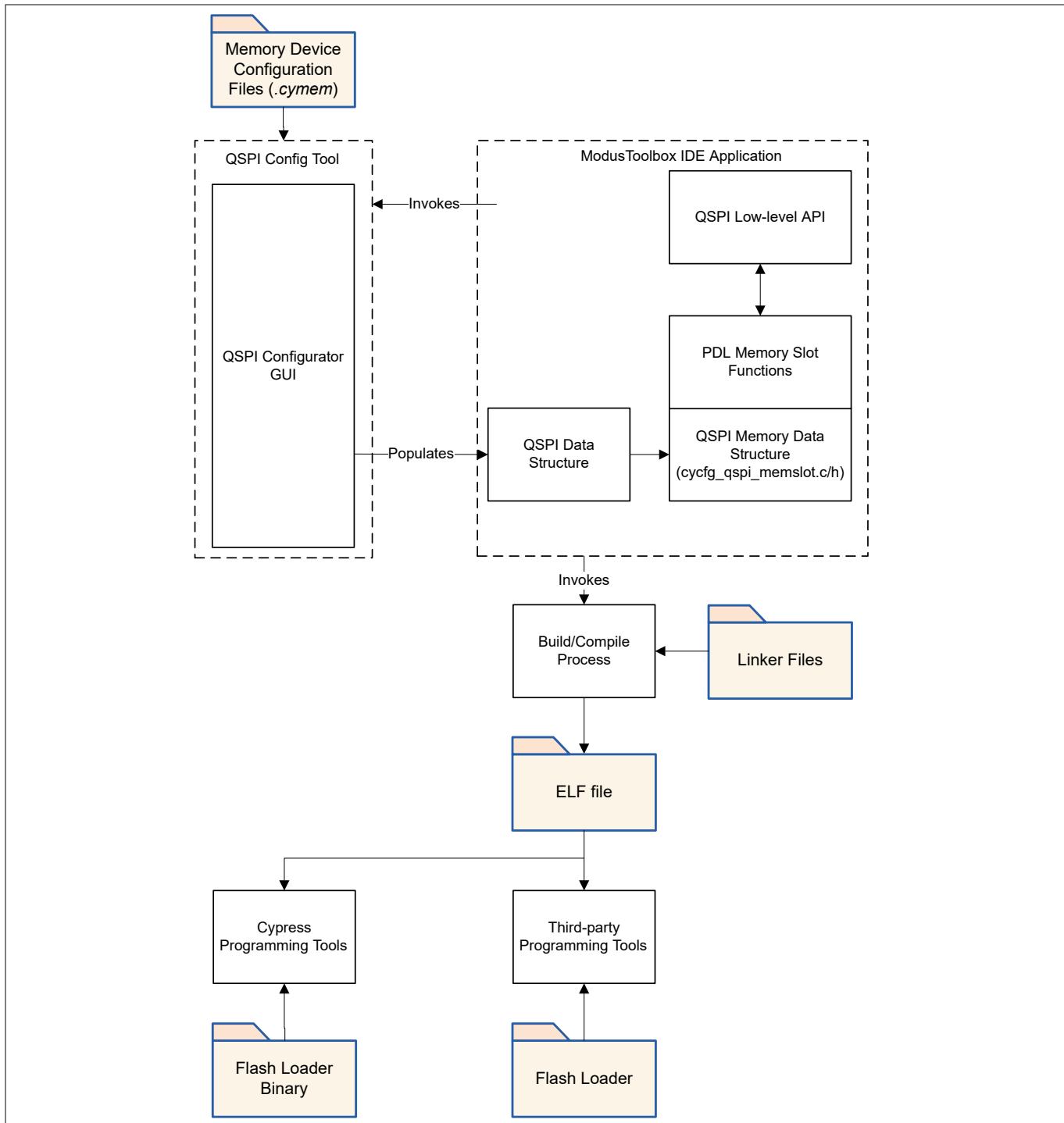


Figure 176

QSPI ecosystem of tools

~~5 PSoC™ 6 application notes~~

~~5.6.4.1 ModusToolbox™ Application Software libraries~~

The ModusToolbox™ Software Environment includes the libraries and files necessary to use QSPI without the need to access the hardware registers directly. The PDL provides low-level configuration for the QSPI block and the API necessary to transfer data to the memory device. The Serial Flash Library and the HAL provide layers of access above the PDL. [Table 22](#) lists the files used by these libraries and the library to which they belong.

Table 22 Source files and libraries

File	Description	Library
cy_smif.c/h	Provides the low-level API for configuring the QSPI hardware and initiating transfers. Used to directly configure the QSPI block.	PDL
cy_smif_memslot.c/h	One level above cy_smif.c. Provides the low-level API for accessing external memory, including status registers and SFDP parameters. Typically used in command mode or for SFDP detection, but also defines memory device status checking functions.	PDL
cyhal_qspi.c/h	Abstracts out any chip specific configuration functions. Functions in the HAL automatically set up the pins used by the block, any required interrupts, and timeouts for each data transfer. Typically used in conjunction with the Serial Flash Library for high-level memory access.	HAL
cy_serial_flash_qspi.c/h	Provides wrappers around functions from both cy_smif.c and cy_smif_memslot.c to ease the use of QSPI. Uses the HAL to set up chip-specific configurations. Functions included in these files limit configurability, but are simple to use and provide the functions needed for most memory accesses.	Serial Flash Library
cy_serial_flash_prog.c	Provides the variables necessary to instruct the programming tools how to program an attached serial flash memory. Depends on files generated in the QSPI Configurator Tool. Typically used with XIP mode to expose external memory addresses for programming.	Serial Flash Library

5.6.4.2 QSPI Configurator tool

The ModusToolbox™ Software Environment includes the QSPI Configurator. The Configurator tool provides a simple graphical interface to set up QSPI to use an external memory device. You can launch the Configurator from within ModusToolbox™ IDE, following step 1 from [Using the Peripheral Driver Library](#). You can also launch the standalone Configurator tool to generate source files that can be used in most IDEs. Navigate to your ModusToolbox™ installation folder and follow this path:

```
{Install Dir}\ModusToolbox\tools_2.0\qspi-configurator\qspi-configurator.exe
```

In the Configurator, you can select your memory device, slot or select line it is connected to, SPI width, memory address ranges, and encryption (for XIP mode). Once you have set your choices, the QSPI Configurator tool automatically generates the source and header files containing the parameters of your external memory device. The parameters of the device are stored in configuration structures, which can be passed as arguments to the PDL or Serial Flash Library functions to easily access your memory device.

5 PSoC™ 6 application notes

DRAFT
Table 23 Source files generated by the QSPI configurator

File	Description
cycfg_qspi_memslot.c/h	Automatically generated from the QSPI Configurator. Provides the definitions for the QSPI Memory device configuration, or sets up default structures for SFDP detection.

The Configurator tool pulls configurations from a database of memory files in XML format, called .cymem files. These files, and an editable template memory file, are typically installed along with the ModusToolbox™ IDE. For more information on these files and how to use the QSPI Configurator Tool, see the [user guide](#).

5.6.4.3 Programming tools

The QSPI block supports programming of external memories through programming tools such as the KitProg3 provided with Infineon kits. For detailed information about how to setup your application to support external memory programming, see [Programming external memory](#).

DRAFT

5 PSoC™ 6 application notes

5.6.5 Configuration

5.6.5.1 QSPI configuration structure architecture

To access an external memory device, you need to make sure your QSPI block is configured correctly for the memory device that you are using. The `cycfg_qspi_memslot.c/h` files generated by the QSPI Configurator tool provides a set of nested structures that contain each configuration parameter for your system. This reduces the amount of time you need to spend on creating command lists and setting up the transfer parameters for each transfer. [Figure 177](#) shows organization of the generated structures.

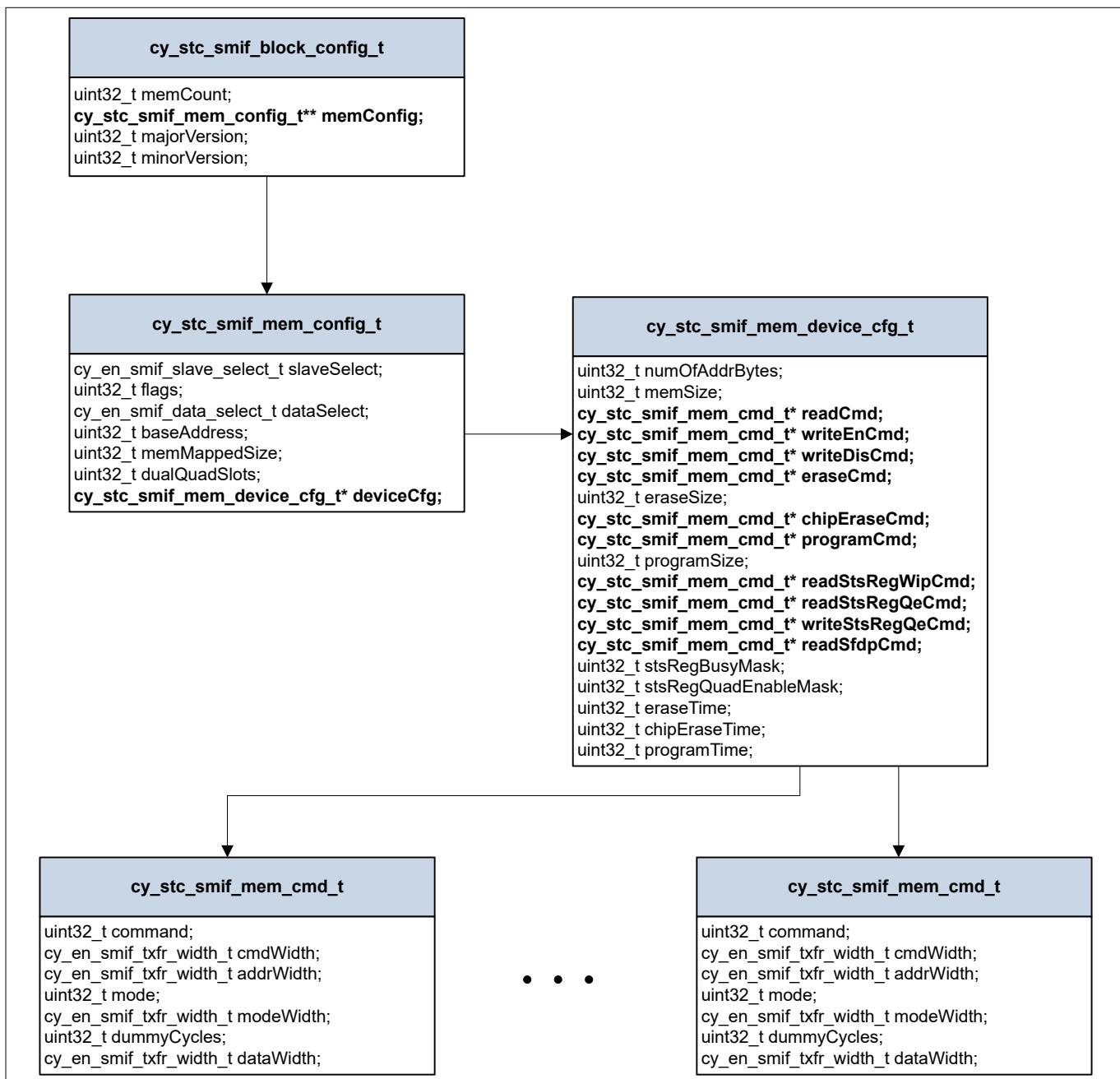


Figure 177 QSPI configuration structure architecture

At the highest level is the `cy_stc_smif_block_config_t` structure. This simple structure contains the number of memory devices connected to the block, the QSPI driver version, and a double pointer to the memory configuration structure.

~~5 PSoC™ 6 application notes~~

Nested within the block configuration structure is the memory configuration structure. This structure contains application-level information about the external memory device, including the SPI slave select slot, the slot of the data lines, the base address, the size of the external address, the number of dual-quad SPI slots, and a pointer to the memory device configuration structures.

The memory device configuration structure, `cy_stc_smif_mem_device_cfg_t`, contains device-specific information including the default memory commands necessary to access the memory device. Typically, the details of this structure can be filled in using information from your memory device's datasheet.

The lowest level structure within the overall QSPI architecture is the memory device command structure, `cy_stc_smif_mem_cmd_t`. For each of the commands listed in the memory device configuration structure, there is a corresponding command structure. These structures specify the requirements of the commands, including command width, address width, mode, mode width, data width, and the number of dummy cycles.

5.6.5.2 Configuration procedure

5.6.5.2.1 SDFP detection

The [serial flash discoverable parameter \(SDFP\)](#) standard provides a set of standard parameter tables that define the capabilities and access specifications for serial flash devices. These tables are internal to serial flash devices that use the SDFP standard and can be read from to determine the settings required to access the device.

The QSPI block supports [SDFP](#) detection. For all devices that support this functionality, it is recommended that you enable SDFP detection to simplify configuration. Using SDFP detection will automatically populate the QSPI configuration structures mentioned in [QSPI configuration structure architecture](#).

You can configure the QSPI block to perform SDFP detection in the QSPI Configurator tool. Follow the steps in [QSPI Configurator](#) and launch the QSPI Configurator tool. From the **Memory Part Number** drop-down menu, select **Auto detect SDFP**, as [Figure 178](#) shows. Make sure your selection is in the slave slot that corresponds to your hardware connection.

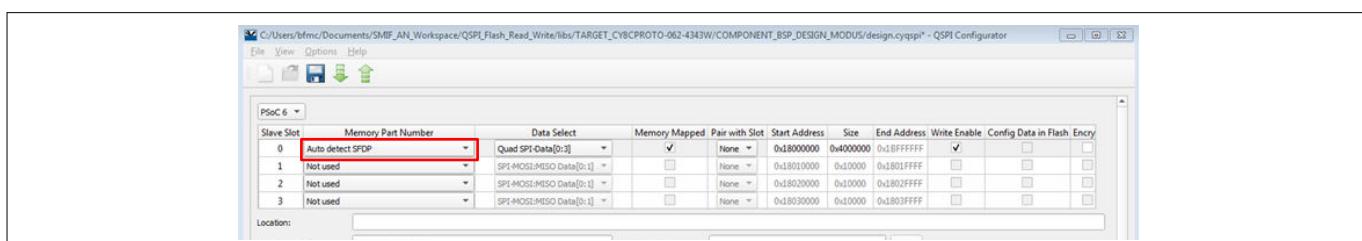


Figure 178 SDFP detection with QSPI configurator

Choosing this option will generate the configuration structures with the prefix "Auto_detect_SDFP", and default values as set in the Configurator tool.

```
cy_stc_smif_mem_config_t Auto_detect_SDFP_SlaveSlot_0
```

In your application, a call to the `cy_SMIF_MemInit` function will perform the SDFP detection and populate the structures with the detected parameters.

5 PSoC™ 6 application notes~~DEAF~~

5.6.5.2.2 Manual configuration

If your device does not support SFDP or to manually configure your device settings, you can select a supported part number from the **Memory Part Number** drop-down menu.

For custom or unsupported devices, you can either create a new memory file (*.cymem) to get support in the Configurator tool, or you can manually populate the structures in the source file (`cycfg_qspi_memslot.c/.h`) with your device's parameters.

To create a new memory file, follow the steps outlined in the QSPI Configurator Guide under the section “Create New Memory File”.

~~5 PSoC™ 6 application notes~~

~~5.6.6 Order of operations~~

When using the PDL or Serial Flash Library, it is important to keep in mind that the functions take care of several important transfer steps automatically. For write transfers, this is typically a two-step process that involves transmitting a write enable command followed by a program command. For read accesses, a read command is usually the only necessary step.

After commands that transfer data, it is important to make sure that the data transfer is complete and the external memory device is ready for the next transaction before using the QSPI block again. This is handled within functions in `cy_smif_memslot.c/h`, however, for lower level PDL accesses, you can use the function `Cy_SMIF_GetTransferStatus()` to determine the current status of the transfer. You can also transmit a device-specific status command using the `Cy_SMIF_TransmitCommand()` function to read the status of the external memory device. To determine the correct read status command for your device, see the device [datasheet](#).

Additionally, many external memory devices require an erase or erase sector operation before writing to the memory device. This can be accomplished by transmitting the device-specific erase command or by using functions provided in the `cy_smif_memslot.c/h` files, the Serial Flash Library files, or the HAL.

The typical flow for a write data transfer follows these steps:

1. Checks that the external memory is not busy
2. Transmits the write enable command
3. Erases the sector of the external memory device that you are going to write to
4. Waits for the external memory erase to finish, this may take a long time. See your memory device [datasheet](#) for erase time specifications.
5. Transmits the write enable command to the external memory device
6. Programs the data into the external memory
7. Waits for the transfer to complete
8. Waits for the external memory device to be ready

A read transfer is simpler, requiring only the read command and a check for the transfer to be complete.

1. Checks that the external memory is not busy
2. Transmits the read command at the address of the data to be read
3. Waits for the transfer to be complete

~~5 PSoC™ 6 application notes~~

~~5.6.7~~ Programming external memory

QSPI supports programming of external memory through programming tools such as the OpenOCD or the KitProg3 device included on PSoC™ 6 MCU kits. To enable this feature, several important steps should be followed or the programming may fail.

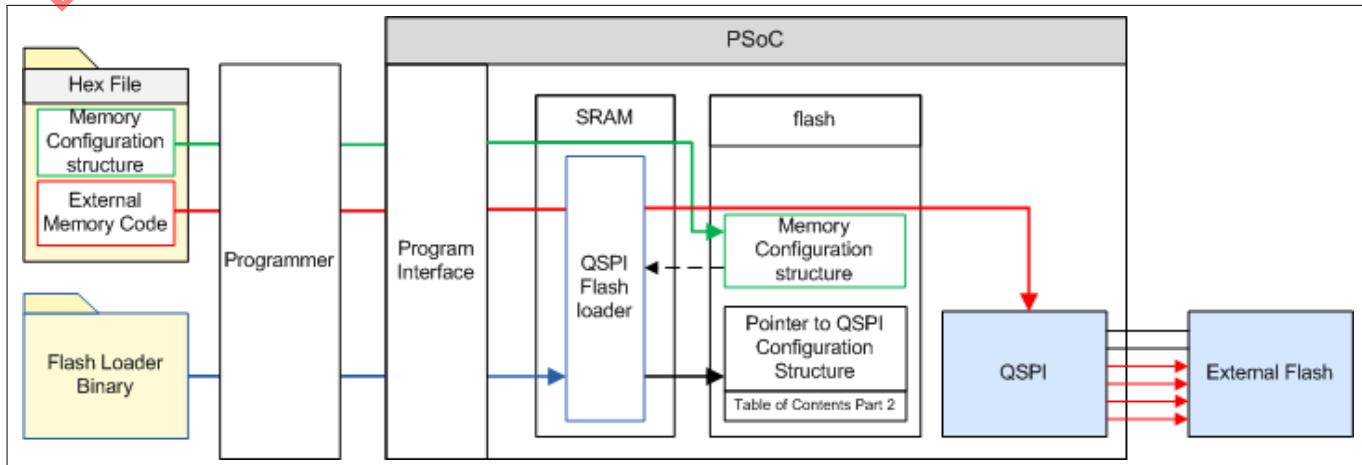


Figure 179 External memory programming flow

In Figure 179, the red arrows represent the code or data that is going to be programmed into the external memory through the QSPI interface. Before the code or data makes its way into the external flash, it must travel through the programmer, the programming interface, the QSPI flash loader, and then finally into the QSPI interface.

At the end of the programming process, the QSPI flash loader is programmed into the PSoC™ 6 MCU SRAM and begins to execute. The flash loader attempts to find a pointer to the QSPI configuration structure from a fixed location in flash as part of the Table of Contents part 2 (TOC2) structure. It is your responsibility to place the pointer in this location so that the flash loader can find it and have the command structure for programming external memory.

To do this, follow these steps:

1. If you are using the Serial Flash Library, navigate to the `cy_serial_flash_prog.c` file and add the definition `#define CY_ENABLE_XIP_PROGRAM`. This automatically includes the necessary pointer in the correct location in flash. No further steps are required.
If you are not using the Serial Flash Library, create a structure which contains a pointer to the `cy_stc_smif_block_config_t` structure and a NULL termination.

```
typedef struct
{
    const cy_stc_smif_block_config_t * smifCfg; /* Pointer to SMIF top-level
configuration */
    const uint32_t null_t; /* NULL termination */
} stc_smif_ipblocks_arr_t;
```

5 PSoC™ 6 application notes

- DRAFT**
2. Create an instance of this structure and place it in a known location.

```
CY_SECTION(".cy_sflash_user_data") __attribute__( (used) )
const stc_smif_ipblocks_arr_t smifIpBlocksArr = {&smifBlockConfig, 0x00000000};
```

3. Place your structure into the TOC2 in the following manner. This structure is a predetermined fixed location in flash known to the flash loader.

```
CY_SECTION(".cy_toc_part2") __attribute__( (used) )
const uint32_t cyToc[128] =
{
    0x200-4,                      /* Offset=0x0000: Object Size, bytes */
    0x01211220,                   /* Offset=0x0004: Magic Number (TOC Part 2, ID) */
    0,                            /* Offset=0x0008: Key Storage Address */
    (int)&smifIpBlocksArr,        /* Offset=0x000C: This points to a null terminated array of SMIF
structures */
    0x10000000u,                  /* Offset=0x0010: App image start address */
    /* Offset=0x0014-0x01F7: Reserved */
    [126] = 0x000002C2,           /* Offset=0x01F8: Bits[1:0] CLOCK_CONFIG(0=8MHz, 1=25MHz,
2=50MHz, 3=100MHz)
                                         Bits[4:2]
    LISTEN_WINDOW(0=20ms,1=10ms,2=1ms,3=0ms,4=100ms)
                                         Bits[6:5] SWJ_PINS_CTL (0/1/3=Disable SWJ,
2=Enable SWJ)
                                         Bits[8:7] APP_AUTHENTICATION (0/2/3=Enable,
1=Disable)
                                         Bits[10:9] FB_BOOTLOADER_CTL: UNUSED */
    [127] = 0x3BB30000           /* Offset=0x01FC: CRC16-CCITT (the upper 2 bytes contain the
CRC and the lower 2 bytes are 0) */
};
```

For more information about the TOC2, its use, and its contents, see the device [architecture TRM](#).

Additionally, the QSPI flash loader does not perform SDFP detection. This means that you must make sure your device is [manually configured](#) and the configuration is stored in flash. To do this using the QSPI Configurator tool, select your memory part from the **Memory Part Number** drop-down menu and make sure the option **Config Data in Flash** is selected.

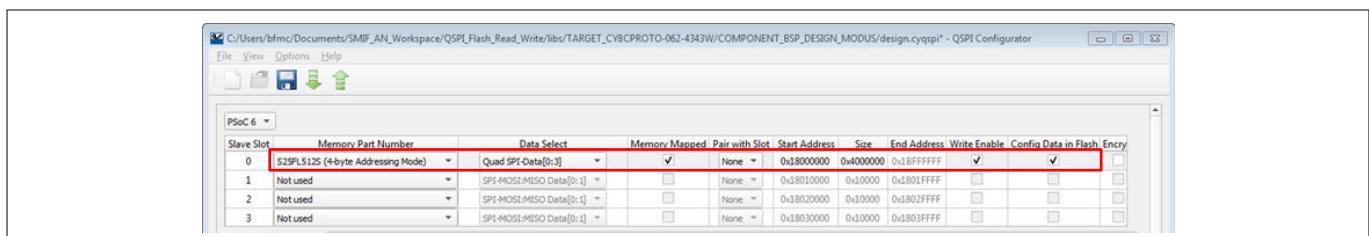


Figure 180 Example configuration for external memory programming

In your firmware, you can then begin placing functions or variables at locations within the external memory addressing range. To do this, you can use the attribute `CY_SECTION(".cy_xip") __attribute__((used))` before a variable or function declaration to place the value in the external memory region.

5 PSoC™ 6 application notes~~DRAFT~~

5.6.8 Security with QSPI

If you are using QSPI to store sensitive data or proprietary libraries, you need to secure your QSPI system. Using the cryptography, explained in [Memory device signal interface](#), is helpful in protecting the data itself, but it does not prevent you from reading out the encrypted data or from injecting other code into an XIP section.

Several important steps are required to secure your QSPI system. The first step is to make sure you are encrypting your data, either through the QSPI provided encryption or through some other robust method. This protects the data itself, making it difficult for a hacker to read sensitive information.

Your next layer of security should come from protecting the QSPI register space from read and write access. This will prevent a hacker from switching the QSPI block mode from XIP to command mode and reading out the encrypted data during runtime. This will also prevent modification of your cryptography key. To do this, make sure your cryptography key is correct and any executable code for the external memory is encrypted and programmed into the external memory device. Then, you can use a Peripheral Protection Unit (PPU) to set the access restrictions for your QSPI block, allowing you to configure read/write restrictions for registers in the QSPI block.

For applications using XIP mode, the code in the external memory also needs to be protected. Because this memory can be shared and accessed by multiple masters (for example, DMA and a CPU) a Shared Memory Protection Unit (SMPU) should be used. The SMPU will allow you to configure the region base address, the size of the memory to protect, and the access restrictions for the memory region. By default, the external memory region is 0x18000000, however, this can be changed in the QSPI Configurator tool.

Finally, make sure that any code stored in the external memory region is validated along with the rest of your application image. For more information on code signing and secure system architecture with PSoC™ 6 MCU, see [AN221111](#).

5 PSoC™ 6 application notes~~DO NOT USE~~
5.6.9 Performance

Serial memory devices are often used as external memory devices for frame buffers or frequently accessed memory like an EEPROM. In these cases, it is important to have low latency transfers with high throughput. The QSPI block follows the SPI protocol for command mode transfers or for transfers in XIP mode where there is a cache miss. In these cases, the latency of a transfer is given by [Equation 1](#).

Interface clock cycles

$$= \left\lceil \frac{\text{opcode size}}{\text{opcode width}} \right\rceil + \left\lceil \frac{\text{address size}}{\text{address width}} \right\rceil + \left\lceil \frac{\text{mode size}}{\text{mode width}} \right\rceil + \text{dummy cycles} + \left\lceil \frac{\text{data size}}{\text{data width}} \right\rceil$$

Equation 1

When encrypting data in command mode, the encryption process takes roughly 13 *clk_hf* cycles, meaning that as long as your calculated transfer time is greater than 13 cycles you will not see a delay.

For XIP mode with caching enabled, any access that hits in the cache will be processed by the cache. Keep in mind that the cache is 16 B and the prefetch buffer is the contiguous 16 B directly following the cache, so every 32 B of contiguous access will incur a single refill for the next 32 B. This refill will behave similarly to a Command mode transfer and can be calculated using the above equation. This makes XIP mode accesses suitable for short read or execute accesses, but for larger read accesses the extra cycles from the repeated refills add an undesirable delay. Thus, for larger data transfers Command mode is advised.

Encryption in XIP mode occurs on-the-fly and does not cause any latency.

5 PSoC™ 6 application notes~~DO NOT USE~~
5.6.10 Summary

This application note explained how to use the PSoC™ 6 MCU QSPI block to access external memory devices. It provided a simple reference flow, explained the features in the block, discussed security design requirements, and performance of the block. For lower-level details about the architecture of the QSPI block within PSoC™ 6 MCU, see the [architecture TRM](#). Additionally, there are many code examples demonstrating how to use external memories with QSPI. See [Related documents](#) for these examples.

5 PSoC™ 6 application notes

Related documents

-
- 6

For a comprehensive list of PSoC™ 6 MCU resources, see [KBA223067](#) in the community. For a comprehensive list of PSoC™ 3, PSoC™ 4, and PSoC™ 5LP resources, see [KBA86521](#) in the community.

Application notes

AN210781 – Getting Started with PSoC™ 6 MCU with Bluetooth Low Energy (BLE) Connectivity	Describes PSoC™ 6 MCU with Bluetooth® Low Energy Connectivity devices and how to build your first PSoC™ Creator project
AN221774 – Getting Started with PSoC™ 6 MCU	This application note helps you explore PSoC™ 6 MCU architecture and development tools, and shows how to create your first project using ModusToolbox™ and PSoC™ Creator
AN215656 – PSoC™ 6 MCU: Dual-CPU System Design	Describes the dual-CPU architecture in PSoC™ MCU, and shows how to build a simple dual-CPU design
AN219434 – Importing PSoC™ Creator Code into an IDE for a PSoC™ 6 MCU Project	Describes how to import the code generated by PSoC™ Creator into your preferred IDE

Code examples

CE220823 – PSoC™ 6 MCU SMIF Memory Write and Read Operation	This example demonstrates the write and read operations to the Serial Memory Interface (SMIF) in PSoC™ 6 MCU
CE222460 – SPI F-RAM Access Using PSoC™ MCU SMIF	CE222460 provides a code example that implements the SPI host controller on PSoC™ 6 MCU using the SMIF Component and demonstrates accessing different features of the SPI F-RAM
CE224073 – SPI F-RAM Access Using PSoC™ 6 MCU SMIF in Memory Mapped (XIP) Mode	SPI F-RAM Access Using PSoC™ 6 MCU SMIF in Memory Mapped (XIP) Mode

Device documentation

[PSoC™ 6 MCU Datasheets](#)

[PSoC™ 6 MCU Technical Reference Manuals](#)

[PSoC™ 6 MCU Programming Specifications](#)

Development kit documentation

[CY8CKIT-062-BLE, PSoC™ 6 BLE Pioneer Kit](#)

[CY8CKIT-062-WIFI-BT, PSoC™ 6 WiFi-BT Pioneer Kit](#)

[CY8CPROTO-062-4343W, PSoC™ 6 WiFi-BT Prototyping Kit](#)

[CY8CPROTO-063-BLE, PSoC™ 6 BLE Prototyping Kit](#)

Tool documentation

ModusToolbox™ IDE	Look in <ModusToolbox install folder>/doc
-----------------------------------	---

5 PSoC™ 6 application notes

5.6.11 Revision history

Document version	Date of release	Description of changes
**	2020-03-10	New application note.
*A	2021-03-08	Updated to Infineon template.
*B	2022-05-16	Updated Figure 171 .

5.7 AN230938 PSoC™ 6 MCU low-power analog

About this document

-
- 7

Scope and purpose

AN230938 explains the low-power analog blocks of the PSoC™ 62 MCU CY8C62x4 product line and provides ways to reduce the power consumption in analog sensing applications.

Intended audience

This application note explains how to reduce the power consumption in the analog blocks of the CY8C62x4 product line. For information on reducing power consumption at the chip level with other peripherals, and to know the power modes of the PSoC™ 6 device in general, see [AN219528 – PSoC™ 6 MCU Low-Power Modes and Power Reduction Techniques](#). This application note also provides basic overview of the analog blocks in the CY8C62x4 product line. For detailed information, refer to the [Technical Reference Manual \(TRM\)](#). If you are new to PSoC™ 6, see [AN228571 – Getting Started with PSoC™ 6 MCU on ModusToolbox™](#).

More code examples? We heard you.

To access an ever-growing list of hundreds of PSoC™ code examples, please visit our [code examples web page](#). You can also explore the video training library [here](#).

~~5 PSoC™ 6 application notes~~

~~5.7.1~~ Introduction

A typical analog front end (AFE) for a measurement system, as Figure 181 shows, involves the use of opamps for filtering and amplification, an analog-to-digital converter (ADC) for digitization, and a CPU for post processing the results. Battery operated systems deploying AFEs pose a challenge in achieving battery life target. In a typical MCU with opamps and ADC, these are significant contributors of power consumption. It becomes critical, specifically in battery-operated systems, to optimize the power levels.

Traditionally, power cycling is used, wherein the MCU is periodically set to low-power state when no measurement is required and is switched to normal power for measurements, resulting in reduced average current. However, in most cases not all peripherals, nor the CPU, need to be active while the ADC is performing the measurements. The CY8C62x4 device achieves significant power savings by allowing only the analog blocks – opamps, ADC, digital-to-analog converter (DAC) and reference to remain ON or power cycle¹⁾ in System Deep Sleep mode and collect the measurement results autonomously while rest of the peripherals, including the CPU, are powered off.

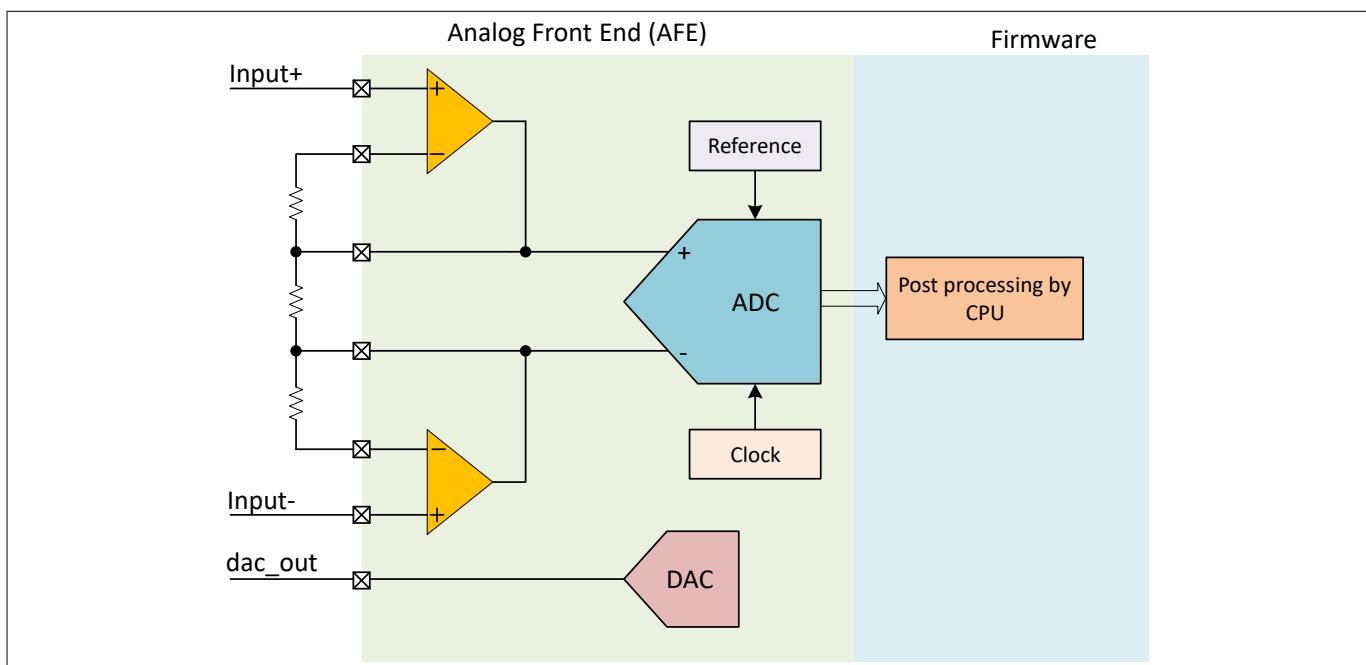


Figure 181 Typical measurement system

The CY8C62x4 product line is unique among the PSoC™ 6 families because it can perform low-power analog sensing. This application note provides an overview of the CY8C62x4 product line and ways to reduce its power consumption. For the evaluation of power consumption, spreadsheet-based calculator can be used from the [AN219528 – PSoC™ 6 MCU low-power modes and power reduction techniques](#).

¹ "duty cycle" term is used for the "power cycle" feature in CY8C62x4 product line.

~~DRAFT~~

5 PSoC™ 6 application notes

5.7.2 Programmable analog features

The CY8C62x4 product line has the following rich set of analog features. For an extensive list of peripherals, see the [device datasheet](#).

- Dual opamps
 - System Deep Sleep operation with configurable duty-cycling
 - Configurable power levels
 - Sample and Hold (S/H) circuit (useful with DAC output buffering)
- Two 12-bit, differential input SAR ADCs
 - Maximum of 2 Msps sample rate in System LP/ULP Power mode and 100 Ksps in System Deep Sleep mode
 - Each ADC allows 16 channels (with 13 unique inputs)
 - Scan of all enabled channels without CPU
 - Reference sources: 1.2V bandgap, V_{DDA} or external
 - Simultaneous sampling
 - Deep Sleep operation with duty-cycling
 - 64-sample FIFO in each ADC
 - Dedicated 16-bit timer for trigger
 - Sensor for die temperature measurement
- 12-bit, 500 Ksps DAC
 - Reference sources: 1.2 V bandgap (buffered through opamp), external input or V_{DDA}
 - Output buffered through opamp
 - Ability to hold output in System Deep Sleep state
- 1.2V bandgap reference
 - Deep Sleep operation
- Dual low-power comparators
 - Ultra Low-power (ULP) mode for operation in System Hibernate mode
 - Interrupt generation to wakeup the device
 - CAPSENSE™ Capacitive Touch
 - Self-capacitance (CSD) and mutual-capacitance sensing (CSX)
 - SmartSense auto-tuning
- Dual analog multiplexer buses
 - Interconnects GPIOs and analog peripherals
 - Ability to split the buses to make multiple connections
- Deep Sleep clock for operating the SAR ADC

Sourced from either 2 MHz Low-Power Oscillator (LPOSC) or Medium Frequency Clock. For the details of each peripheral, see the [Technical Reference Manual](#).

~~DRAFT~~ 5 PSoC™ 6 application notes

5.7.2.1 Differences with other PSoC™ 6 devices

Table 24 provides a comparison of CY8C62x4 with other PSoC™ devices.

Table 24 Comparison of CY8C62x4 with other PSoC™ 6 devices

Feature	CY8C62x4	Other PSoC™ 6 devices
Number of SAR ADCs	2	1
Maximum ADC sample rate	2 Msps in system LP/ULP mode and 100 Ksps in System Deep Sleep mode	1 Msps/2 Msps ²⁾
SAR ADC in System Deep Sleep mode	✓	✗
SAR ADC FIFO	✓	✗
Dedicated timer for SAR ADC trigger	✓ ³⁾	✗
Low-power Oscillator (LPOSC)	✓	✗
Configurable number of ADC scans per trigger	✓	One scan per trigger or continuous scan
Analog reference in System Deep Sleep mode	✓	✓
Opamp in System Deep Sleep mode	Always ON or power cycle	Always ON
DAC in System Deep Sleep mode ⁴⁾	✓	✓

For more details on low-power operation of analog blocks, see [Low-power mode operation of analog resources](#) section.

5.7.2 Analog routing

Along with the rich set of analog blocks, CY8C62x4 provides flexible routing to implement circuit configurations that can be dynamically controlled. [Figure 182](#) illustrates the analog routing diagram.

SARMUX: SARMUX is used by the SAR ADC to connect to different sources such as GPIOs, opamp outputs, and temperature sensor. There are two SARMUX blocks in the device, one for each ADC.

SARBUS: SARBUS is used by the SAR ADC to connect to Opamps via SARMUX. There are two SARBUS blocks: SARBUS0 and SARBUS1.

Analog Multiplexer Bus (AMUXBUS): AMUXBUS is used to interconnect GPIOs from certain ports and to route signals to/from the peripherals. There are two analog mux buses in the device: AMUXBUSA and AMUXBUSB. These buses can be split to create multiple segments which helps to route more than two signals. Note that AMUXBUS connections are not available in Deep Sleep and Hibernate modes. If Deep Sleep or Hibernate operation is required, the Low-power comparator must be connected to the dedicated pins. This restriction also includes routing of any internally-generated signal, which uses the AMUXBUS for the connection.

Dedicated port pins for the peripherals provide slight advantage over other pins in terms of routing resistance and capacitance. See [AN218241 – PSoC™ 6 MCU Hardware Design Considerations](#) for the recommended pins.

² Maximum sample rate depends on the device family.

³ One timer for both the SAR ADCs.

⁴ DAC output held constant at a configured value in system Deep Sleep mode.

5 PSoC™ 6 application notes

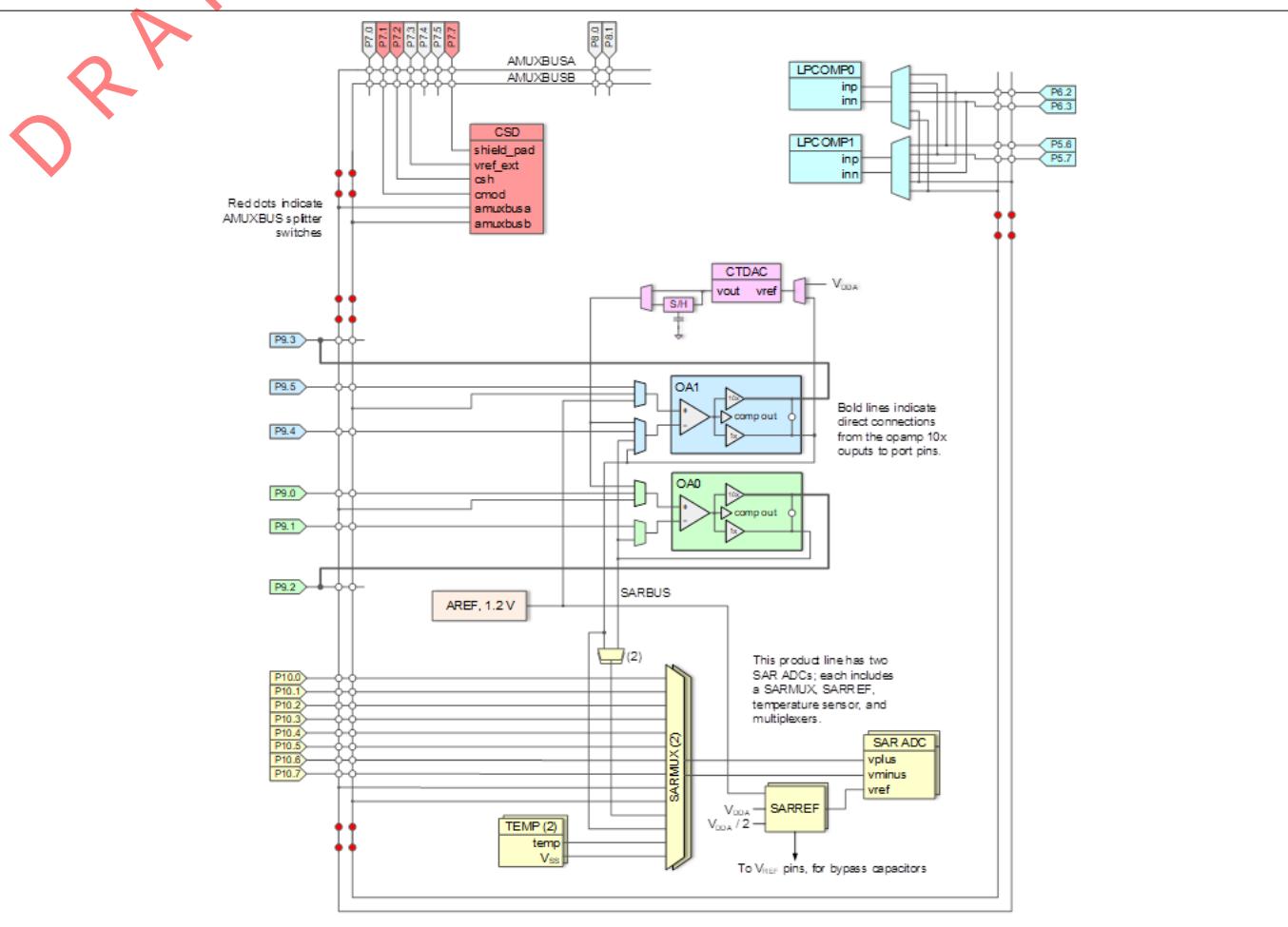


Figure 182 Analog routing in CY8C62x4

~~5 PSoC™ 6 application notes~~

~~DRAFT~~ 5.7.3 Low-power mode operation of analog resources

This section describes Deep Sleep operation of analog resources in detail. [Table 25](#) provides the operational capability in different low-power modes.

Table 25 Operation in low-power modes

Analog resource	System LP/ULP	System deep sleep	System hibernate
SAR ADCs	✓	✓	✗
Opamps	✓	✓	✗
Analog reference (AREF)	✓	✓	✗
DAC	✓	✓ ⁵⁾	✗
Low-power comparators	✓	✓	✓
CAPSENSE™	✓	✗	✗
AMUXBUS switches	✓	✗	✗

If you are not familiar with the low-power modes of PSoC™ 6, see [AN219528 – PSoC™ 6 MCU Low-Power Modes and Power Reduction Techniques](#).

5.7.3.1 SAR ADCs

SAR ADCs are supported with resources to operate autonomously in System Deep Sleep mode. These resources include the analog reference generator, LPOSC or medium-frequency clock for running both the SAR ADCs, timer for trigger (common for both ADCs), and a FIFO for storing the data.

The ADCs operate in one of the following two modes:

- Always-ON in LP/ULP mode and powered-OFF in System Deep Sleep mode
- Always-ON in LP/ULP mode and duty-cycling in System Deep Sleep mode

[Figure 183](#) illustrates the operation of an ADC in System Deep Sleep mode and the transition from or to LP/ULP mode.

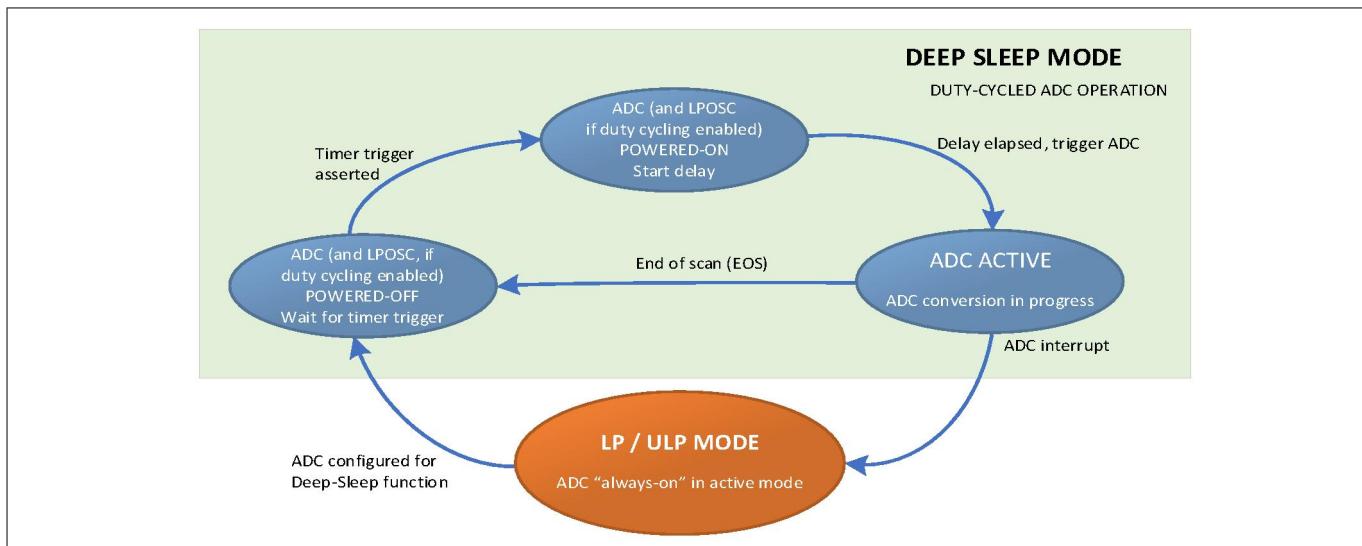


Figure 183 Operation of ADC

⁵ DAC output can only be held at a configured value in System Deep Sleep.

5 PSoC™ 6 application notes

~~CONFIDENTIAL~~

In the LP/ULP mode, the ADC is in an always-on state along with the supporting resources. In the System Deep Sleep mode, the ADC remains powered-off until timer trigger is received. The timer is the only option for an ADC trigger in System Deep Sleep mode. Upon receiving a trigger, a programmable delay (called as power-up delay) is initiated to allow the ADC, the analog reference⁶⁾ and LPOSC clock⁶⁾ to settle before the conversion begins. After this delay, conversion of all enabled channels begins (which corresponds to one scan) and the results are stored in the FIFO. Power for the ADC is then turned OFF and it waits for the next timer trigger.

If both ADCs are used in System Deep Sleep mode, power is not turned OFF until results from both ADCs are moved to their respective FIFOs.

Interrupts are enabled to allow the ADC to wake up the device (see section [Interrupts](#)).

Figure 184 illustrates the timing diagram of SAR ADC duty-cycling in System Deep Sleep mode and the resulting average current consumption.

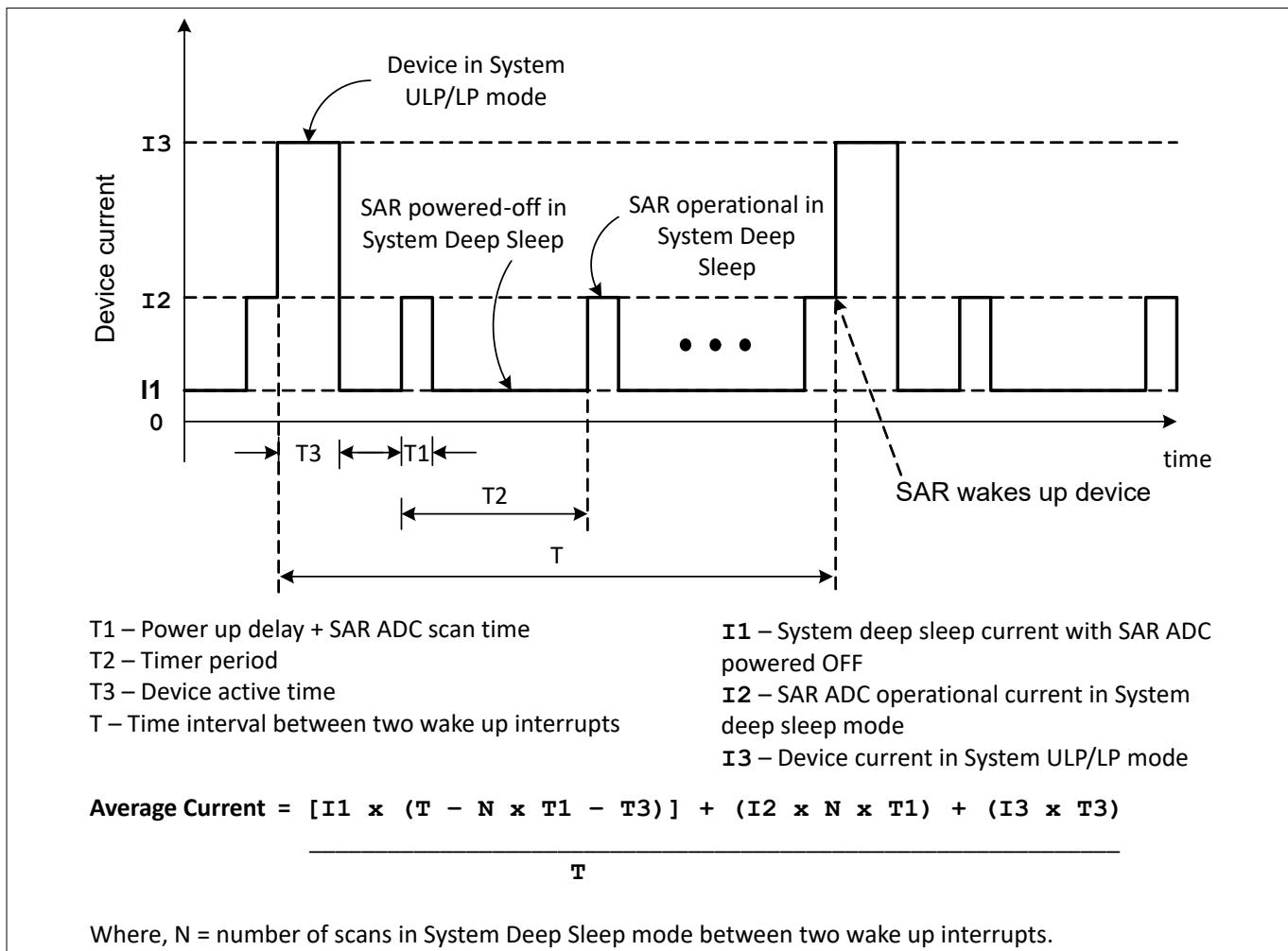


Figure 184 Timing diagram for ADC duty cycling

⁶⁾ Settling time is required for the supporting resources if these resources are duty-cycled along with the ADC. The analog reference generator remains powered-on if CTBm does not use duty-cycling setting (see [Opamp](#) section for details). LPOSC duty-cycling is optional and it can remain ON continuously in System Deep Sleep mode.

~~5 PSoC™ 6 application notes~~

~~5.7.3.1.1~~ Clock source and limitations

For operating in System Deep Sleep mode, SAR ADC should be clocked from either LPOSC or Medium Frequency Clock. The same clock can be used while operating in System LP/ULP mode. But, using LPOSC or Medium Frequency Clock brings in certain limitation as [Table 26](#) shows.

Table 26 Limitations in SAR ADC based on clock selection

Feature	Peripheral clock	LPOSC or medium frequency clock
Applicable device operating mode	System LP/ULP mode	System LP/ULP mode and System Deep Sleep mode
Maximum clock frequency	36v MHz	2 MHz
Trigger source	Firmware, GPIOs, TCPWMs, Timer or LPCOMP	Timer
Continuous conversion	Available	Not available
Interleaved averaging	Available	Not available
Result register	Available	Not available (only FIFO)
Collision and overflow interrupt ⁷	Available	Not available
Injection channel conversion and associated interrupts ⁷	Available	Not available

5.7.3.1.2 Input options

In System Deep Sleep mode, the ADC can receive the inputs from the opamp outputs and from SARMUX port pins. Inputs from other port pins are not available because the AMUXBUS switches do not work in System Deep Sleep mode.

5.7.3.1.3 Interrupts

The following interrupts are available to wake up the device from System Deep Sleep mode or when LPOSC/Medium Frequency Clock is used:

- End of Scan (EOS) – Interrupt is generated at the end of conversion of all the enabled channels or after a “Scan Count” number of scans. Scan Count is specified to execute specific number of scans per trigger
- SAR Range Detection – Interrupt is generated when the channel result satisfies the configured condition
- SAR Saturation – Interrupt is generated when the conversion result (before averaging, if enabled) is equal to the minimum or maximum value. This interrupt can individually be enabled for each channel
- FIFO Level – Interrupt is generated when the number of data samples in the FIFO exceeds the configured “Level” value
- FIFO Overflow – Interrupt is generated when new data sample is posted into an already full FIFO
- FIFO Underflow – Interrupt is generated when the CPU or the DMA tries to read an empty FIFO

For a complete list of interrupts, see the ADC chapter in the [Technical Reference Manual](#).

⁷ See [Technical Reference Manual](#) of the device for the details.

~~DEAF~~ 5 PSoC™ 6 application notes

5.7.3.2 Opamp

Similar to the ADCs, the opamps can also work in System Deep Sleep mode, in the following modes:

- Always-ON in LP/ULP mode and powered-OFF in System Deep Sleep mode
- Always-ON in LP/ULP mode and duty-cycling in System Deep Sleep mode
- Always-ON in LP/ULP mode and System Deep Sleep mode

Note that the ADC does not have a mode (c) available. Opamp mode (c) is useful to avoid wake up times which could be longer with larger external passive components in the opamp feedback path.

The duty-cycling of opamps is linked to the duty-cycling of the ADCs in System Deep Sleep mode, as [Figure 185](#) shows. Duty-cycling of opamps is possible only when at least one of the ADCs is used in System Deep Sleep mode. The timer trigger wakes up both the opamps and the enabled ADCs. When the conversions are complete, the opamps and ADCs are powered-OFF at the same time. When opamp duty-cycling is not used⁸ or if the ADCs are configured to operate only in LP/ULP mode, the opamps remain ON continuously throughout the System Deep Sleep time.

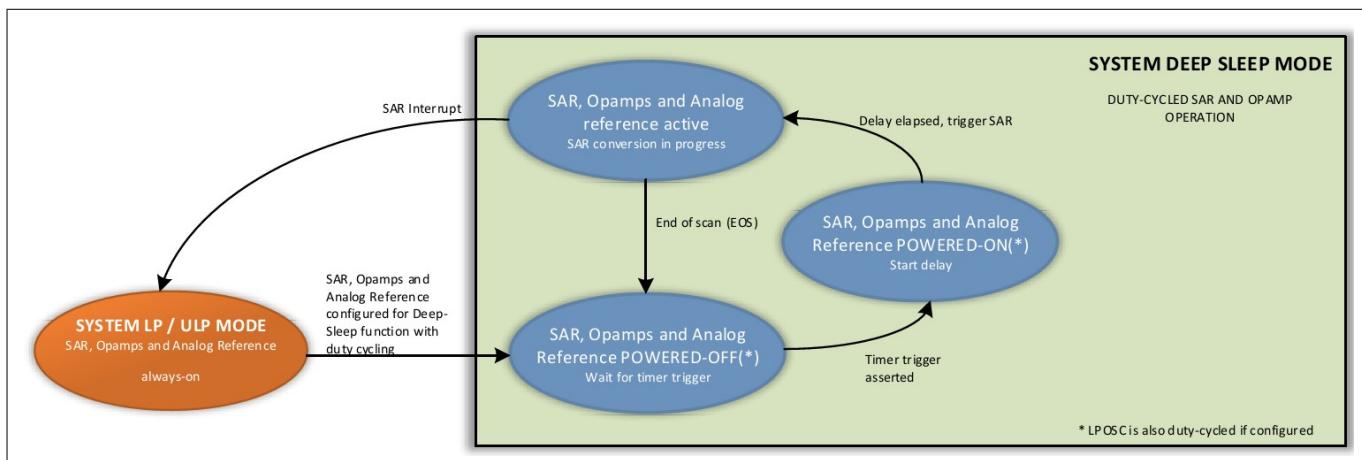


Figure 185 **Duty-cycled SAR ADC and opamps**

Analog reference is used by both the SAR ADCs and the opamps; charge pump is used by Opamps to boost its input range. Their state in System Deep Sleep mode depends on the settings of SAR ADCs and Opamps as [Table 27](#) shows.

Table 27 **Analog reference and charge pump in system deep sleep mode**

Opamps in system deep sleep mode	Analog reference	Charge pump
Powered-OFF	Always-ON if atleast one of the SAR ADCs is configured for operation in System Deep Sleep mode	OFF
Duty-cycle	Duty-cycles	Duty-cycles
Always-ON	Always-ON	OFF

Note: It is not possible to keep one Opamp in Duty-cycle mode and another Opamp in Always-ON mode.

⁸ Duty-cycling is disabled for pamps when peripheral clock is selected for the charge pump circuit. To use opamps in duty-cycling mode, select deep sleep clock as clock source for opamps.

~~5 PSoC™ 6 application notes~~

~~5.7.3.2.1~~ Power modes

Each opamp has three power settings: high, medium and low; there is additional option to select a 100 nA or a 1 μ A current reference from the Analog Reference block which is common for all the opamps in the device. These give a total of six configurable power options for the opamps. [Table 28](#) provides data for the current consumption and gain-bandwidth product while operating the opamp in System Deep Sleep mode.

Table 28 Opamp in system deep sleep mode

Opamp power mode	AREF current ⁹⁾	Current consumption – typical (μ A)	Gain-BW product typical (MHz)
High	1 μ A	1300	4
Medium	1 μ A	460	2
Low	1 μ A	230	0.5
High	0.1 μ A	100	0.5
Medium	0.1 μ A	40	0.2
Low	0.1 μ A	15	0.1

Higher the power setting, higher the gain-bandwidth product and higher the current consumption for a given AREF current. Power setting also affects other opamp parameters. Refer the [device datasheet](#) for details.

5.7.3.3 DAC

DAC, in System Deep Sleep mode, can hold its last configured value. It is not possible to change this value during System Deep Sleep mode. This voltage is buffered using CTBm opamp and driven to a pin. This is useful in applications, where voltage is continuously required for biasing.

There is another useful feature of duty-cycling the DAC which works in conjunction with S/H circuit of CTBm. In this method, firmware connects DAC output to Opamp input in order to sample the DAC output and then DAC is disconnected and powered down. Opamp input is held at a certain potential which is periodically refreshed using firmware. [Figure 186](#) shows a simplified diagram of S/H between DAC and opamp. For the detailed configuration, refer the [Technical Reference Manual](#).

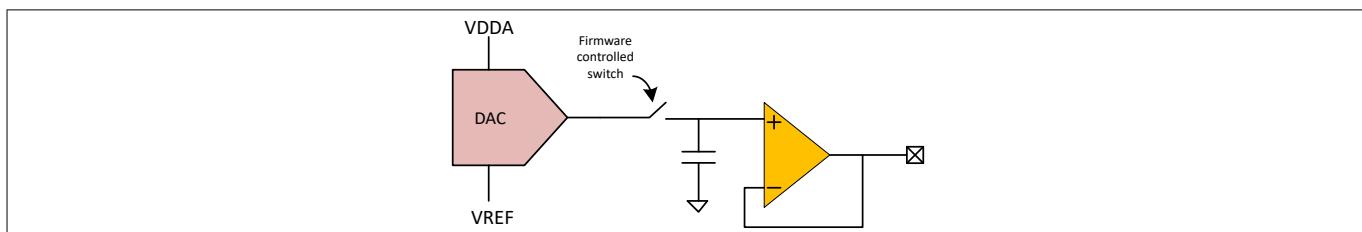


Figure 186

DAC - opamp S/H

⁹ Note that the selected AREF current is common for all the opamps in the device.

~~5 PSoC™ 6 application notes~~

~~5.7.3.4~~ Analog reference (AREF)

The AREF block generates voltage and current references required for SAR ADCs, Opamps, DAC and CAPSENSE™. This block can operate in both System LP/ULP and System Deep Sleep modes. In the System Deep Sleep mode, AREF block can be configured in different modes as [Table 29](#) shows.

Table 29 AREF in system deep sleep mode

AREF setting	Details
Disabled	OFF
Fast wakeup	This mode is useful for fast startup from disabled state to active state while transitioning from System Deep Sleep mode to System LP/ULP mode. Reference voltage and current are not available in System Deep Sleep mode.
Opamp reference current enabled	In this mode, AREF provides reference current to the CTBm Opamps. No voltage reference is available.
All references enabled	In this mode, voltage and current references are available. As explained in the Opamp section, AREF is required for opamp to operate in Deep Sleep mode, therefore AREF also needs to duty-cycle if the opamp is configured to duty-cycle in System Deep Sleep mode.

AREF current, available to the CTBm opamps, can be configured to either 100 nA or 1 µA which affects the power consumption (see [Table 28](#)).

5.7.3.5 Low-power comparator (LPCOMP)

LPCOMP is the only analog block capable of operating even in Hibernate mode. There are two LPCOMPs in the device. It provides an option to generate interrupt so as to wake up the device. The block provides three programmable power levels: normal, low, and ultra-low. The response time of the block increases with the decrease in the power level as [Table 30](#) shows.

Table 30 LPCOMP modes

LPCOMP power mode	Current consumption – maximum (µA)	Response time – maximum (µs)
Normal	150	0.1
Low	10	1
Ultra-Low	0.85	20

Power mode selection also impacts other parameters of the block. Refer the [device datasheet](#) for the specifications.

5 PSoC™ 6 application notes

~~DRAFT~~ 5.7.4 Power optimization techniques

This section provides different ways to reduce the power consumption while using analog blocks of the device.

~~DRAFT~~ 5.7.4.1 Power modes of the block

Opamps and LPCOMPs provide configurable power options. Depending on the application requirement, block power can be optimized. Note that lower power levels come with a penalty on the performance. Refer the [Opamp](#) section and [Low-power comparator \(LPCOMP\)](#) section for details.

5.7.4.2 Duty-cycling the blocks

Many applications do not require continuous scanning of analog inputs and the CPU in its operation in the field. In such cases, duty-cycling of analog sources in System Deep Sleep mode yields significant power savings. Following sections describe three scenarios with the use of duty-cycling for SAR ADC and Opamp. Bench current measurement is done using the [CY8CKIT-062S4 PSoC™ 62S4 Pioneer Kit](#).

5.7.4.2.1 Example: SAR ADC

Consider an application which uses one SAR ADC. [Table 31](#) and [Figure 187](#) provides power consumption results for the device for two cases – when the SAR ADC is operated in System ULP mode and when duty-cycling is used in System Deep Sleep mode.

Table 31 Example - device power consumption with SAR ADC

Device power mode	Average device current (mA)	Conditions
System ULP mode (CPU Active)	4.59	CPU is in active state executing loop instruction. SAR ADC is always powered-ON, sampling periodically at 1 kHz with the scan time of 1 ms.
System ULP mode (CPU Sleep)	2.87	CPU is set to sleep, SAR ADC is always powered-ON, sampling periodically at 1 kHz with the scan time of 1 ms.
System Deep Sleep mode	0.17	SAR ADC power is duty-cycled and it samples periodically at 1 kHz with the scan time of 1 ms. Interrupt is disabled to keep the device in System Deep Sleep.

5 PSoC™ 6 application notes

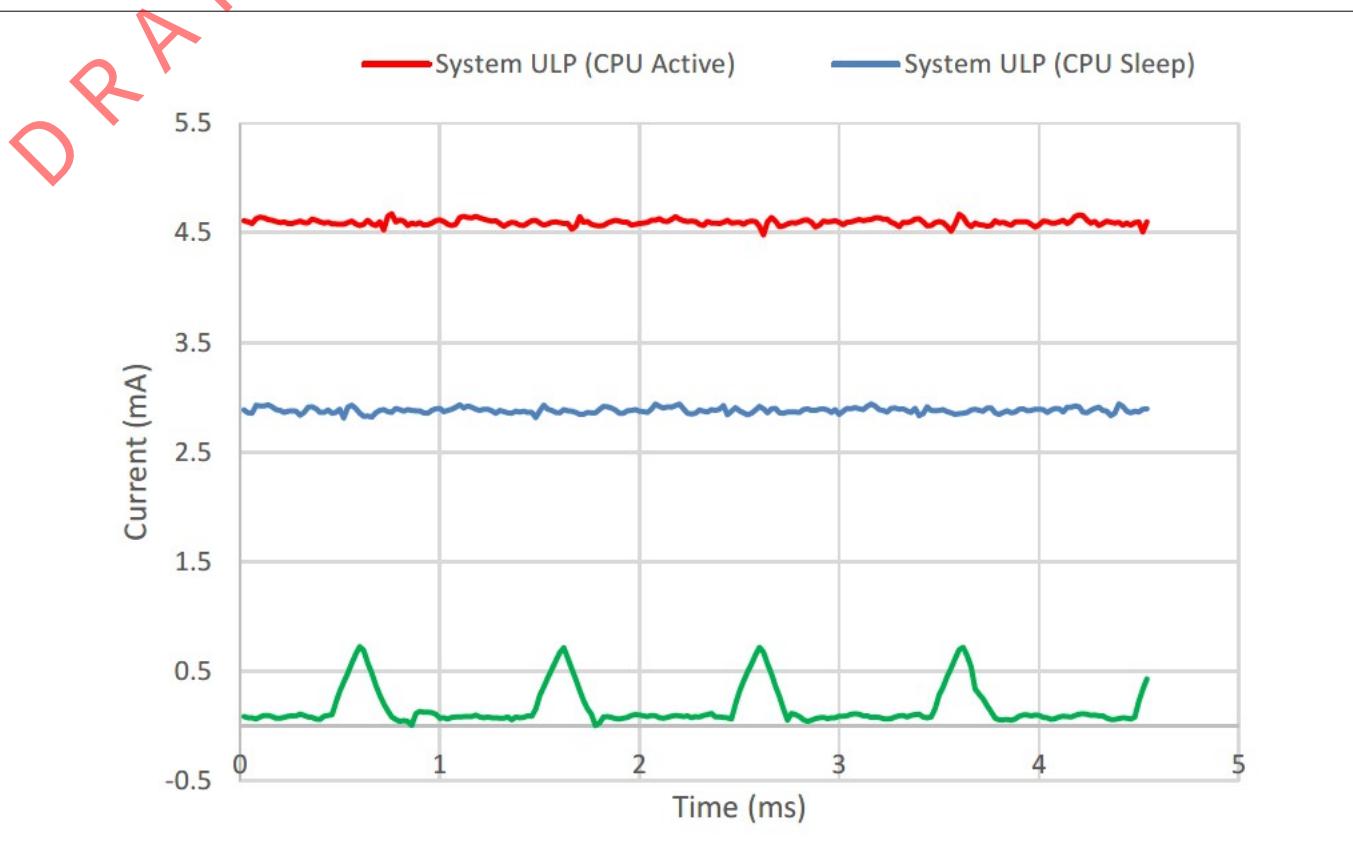


Figure 187 SAR in system ULP and deep sleep modes

The current measurement in Figure 187 setup uses the following key settings:

- Core Regulator: ULP, Normal Current LDO
- PLL Frequency: 50 MHz, FLL: Disabled
- CM4 Frequency: 50 MHz
- V_{DDA} : 3.3 V

SAR is configured with following settings:

- SAR Clock: Duty-cycled LPOSC 2 MHz
- Vref: V_{DDA}
- Acquisition time: 1 μ s
- Number of channels: 1
- Timer frequency/period: 1 kHz/1 ms
- Timer clock input: ILO
- Averaging: Enabled- 16 samples
- Power-up delay: 20 μ s
- Calculated conversion time (scan time): 136 μ s

5.7.4.2.2 Example: SAR ADC + opamp

For the application with opamp buffer output connected to SAR ADC, Table 32 and Figure 188 provides the bench current measurement results. Three uses cases demonstrated: both the SAR ADC and opamp operated in System ULP mode without duty-cycling, with duty-cycled SAR ADC and always-ON Opamp in System Deep Sleep mode and with duty-cycled SAR ADC and Opamp in System Deep Sleep mode.

~~DRAFT~~

5 PSoC™ 6 application notes

Table 32 Example – device power consumption with SAR ADC and opamp

Device power mode	Average device current (mA)	Conditions
System ULP mode (CPU is active)	5.7	CPU is in active state executing loop instruction. SAR ADC and opamp are always powered-ON, SAR sampling the opamp output periodically at 1 kHz with the scan time of 1 ms. In this case, opamp charge pump is enabled with peripheral clock.
System ULP mode (CPU in sleep)	4.03	CPU is set to sleep, SAR ADC and opamp are always powered-ON, SAR samples the opamp output periodically at 1 kHz with the scan time of 1 ms. In this case, the opamp charge pump is enabled with peripheral clock.
System Deep Sleep mode with Opamp always-ON	1.32	SAR ADC power is duty-cycled and opamp is always ON. SAR ADC samples periodically at 1 kHz with the scan time of 1 ms. Interrupt is disabled; hence CPU never gets woken up. In this case, the opamp charge pump is disabled, resulting in reduced input range.
System Deep Sleep mode with Opamp duty-cycled	0.32	SAR ADC and opamp power are duty-cycled. SAR ADC samples periodically at 1 kHz with the scan time of 1 ms. Interrupt is disabled to keep the device in System Deep Sleep mode. In this case, the opamp charge pump is enabled with LPOSC clock.

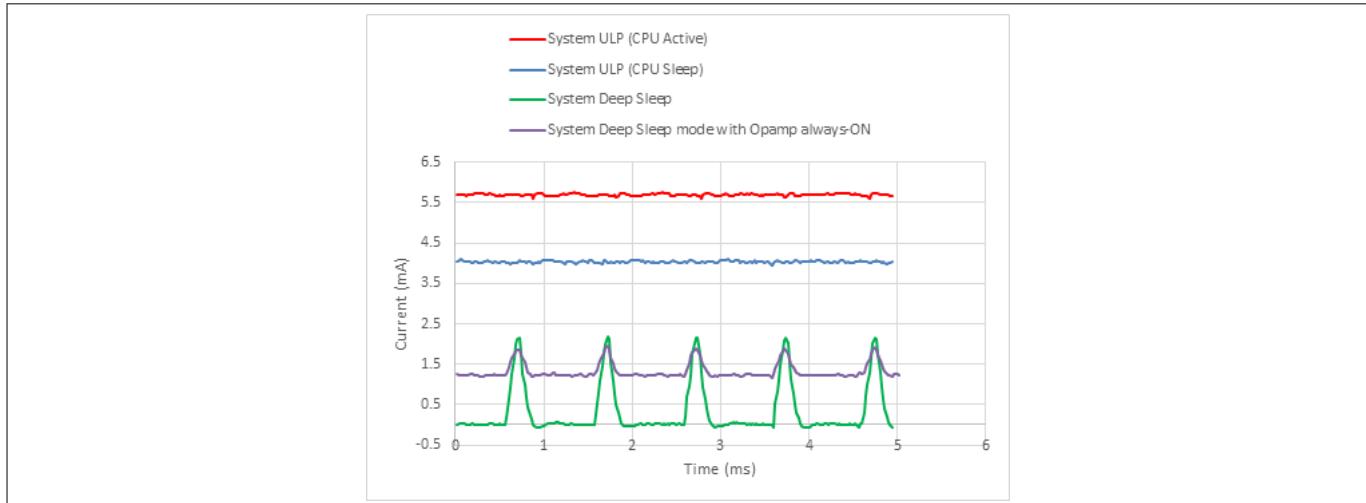


Figure 188 SAR ADC and opamp in system ULP and deep sleep modes

The current measurement in [Figure 188](#) setup uses the following key settings:

- Core Regulator: ULP, Normal Current LDO
- PLL Frequency: 50 MHz, FLL: Disabled
- CM4 Frequency: 50 MHz
- V_{DDA} : 3.3 V

SAR is configured with following settings:

- SAR Clock: Duty-cycled LPOSC 2 MHz
- V_{ref} : V_{DDA}
- Acquisition time: 1 μ s

5 PSoC™ 6 application notes

- ~~DRAFT~~
- Number of channels: 1
 - Timer frequency/period: 1 kHz/1 ms
 - Timer clock input: ILO
 - Averaging: Enabled- 16 samples
 - Power-up delay: 20 μ s¹⁰
 - Calculated conversion time (scan time): 136 μ s

Opamp is configured with following settings:

- Follower configuration
- Power mode: High
- AREF current: 1 μ A
- Pump clock: 24 MHz System Resources Pump Clock or 2 MHz LPOSC clock

The [Low Power Analog Front- End using OpAmp and SAR ADC](#) code example available on GitHub can be used as reference to get started with configuring and programming for low power designs.

5.7.4.3 SAR ADC scan frequency

One scan of SAR ADC involves sampling and conversion of all the enabled channels. When duty-cycling is used, power savings depend on the interval between the two scans triggered by the Timer. Longer the interval, that is lower the scan frequency, lower the power consumption. For applications where the signal to be measured vary slowly, the scan interval should be set longer.

Based on the test settings provided in [Example: SAR ADC](#) section (except for the timer clock input; here, WCO is used instead of ILO for better accuracy), [Figure 189](#) shows current numbers with the bench testing with [CY8CKIT-062S4 PSoC™ 62S4 Pioneer Kit](#) for different scan frequencies and average sample count while operating in System Deep Sleep mode.

¹⁰ Note that if the opamp circuit uses external passive components, additional delay should be included to account for the opamp output settling.

5 PSoC™ 6 application notes

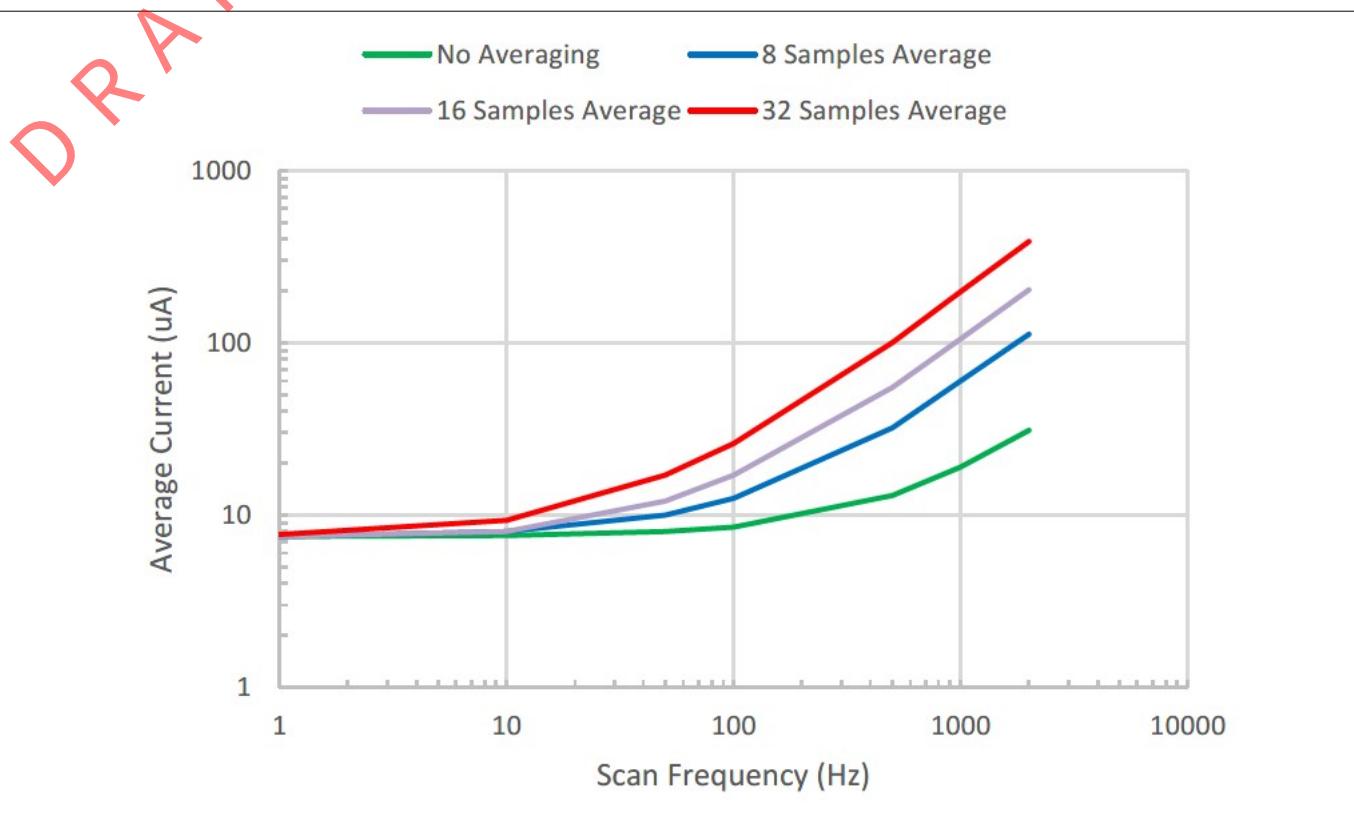


Figure 189 Device current vs SAR ADC scan frequency

5.7.4.4 SAR ADC acquisition time

Acquisition time of SAR ADC channels should be set to minimum possible value to reduce the operating time of SAR ADC and power consumption. Acquisition time is selected from one of the four programmable 10-bit fields. If the SAR ADC clock is set to peripheral clock of 36 MHz (maximum frequency), the acquisition time can be set from 83 ns (three clocks) to 28 μ s. With 2 MHz LPOSC or Medium Frequency Clock, the acquisition time can be set from 0.5 μ s (1 clock) to 512 μ s. Selection of the acquisition time depends on the source resistance, input resistance, and input capacitance of SAR ADC.

Figure 190 shows the equivalent diagram of the SAR ADC input. When the acquisition begins with the closing of S_{ACQ} , the C_{HOLD} charges towards the input signal value. For 12-bit resolution, it takes 9RC time to charge within $\frac{1}{2}$ LSB of the final value. Therefore, the minimum acquisition time is as shown below.

$$T_{ACQ} \geq 9 \times R \times C_{HOLD}$$

where,

$$R = R_{SRC} + R_{SW1} + R_{SW2}$$

R_{SRC} = Source resistance

R_{SW1} = SAR sampling switch resistance

R_{SW2} = Routing resistance

$C = C_{HOLD}$ is the SAR input capacitance and it is typically 5pF

5 PSoC™ 6 application notes

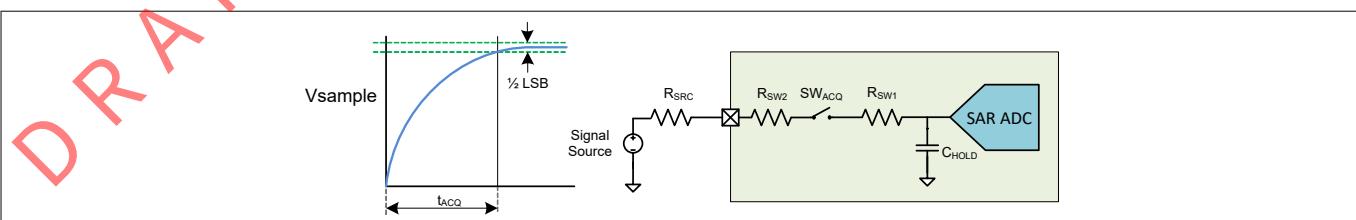


Figure 190 SAR acquisition time

If the input is taken from SARMUX port pins, $R_{SW1} + R_{SW2}$ is typically $1\text{ k}\Omega$. If routing involves SARBUS or AMUX bus, additional resistance should be included which results in increased acquisition time requirement. It is therefore recommended to use SARMUX port pins for SAR ADC inputs.

Note: *If the averaging is enabled, there will be a multiplying effect on the acquisition time selected. For example, if the averaging count is 16 and the acquisition time is 1 μs more than what is required, then it would cause wastage of power in 16 μs of time. For example, if the total amount of time spent is 152 μs out of 1 ms instead of 136 μs as shown in Example: SAR ADC section, a loss of approximately 12% more power occurs.*

5.7.4.5 SAR ADC reference buffer

Each SAR ADC includes a reference buffer to drive its reference. The input options to the reference buffer are the 1.2 V from Analog Reference block and $V_{DDA}/2$. The other options (V_{DDA} and external reference from the GPIO) are connected directly to the reference terminal of SAR ADC and the reference buffer is disabled. Thus, power can be saved if V_{DDA} or external reference is used. Refer the [device datasheet](#) for the current consumption with internal and external reference specifications.

Reference buffer power can be adjusted based on the SAR ADC clock frequency. ModusToolbox™ software automatically configures the buffer power based on the selected clock frequency. If SAR ADC is configured manually, then buffer power should be configured to 100% when operating above 18 MHz, 60% when operating at 3.6 MHz to 18 MHz, and 30% when operating below 3.6 MHz. Refer the [Technical Reference Manual](#) for more details.

The reference buffer output can be routed to pin for filtering using a bypass capacitor¹¹). Start up time of the SAR ADC block varies with the capacitor used for the internal reference voltage filtering as [Table 33](#) shows.

Table 33 Startup times

Reference and Bypass Capacitor	Maximum Start Up Time (μs)
Internal reference without bypass capacitor, external or V_{DDA} reference	10
Internal reference with 100 nF	210

¹¹ Internal reference voltage filtering using bypass capacitor is required for operating the SAR ADC at 18 MHz maximum frequency (i.e. for 1 Msps sample rate). If not used, the maximum clock frequency can be 3.6 MHz (that is, 200 Ksps sample rate). For frequencies higher than 18 MHz and up to 36 MHz, VDDA or external reference should be used; reference buffer cannot be used. Also, for using clock frequencies higher than 18 MHz, VDDA should be minimum 2.7 V. see the [device datasheet](#) for specifications.

5 PSoC™ 6 application notes~~DO NOT USE~~
5.7.4.6 LPOSC or medium frequency clock

SAR ADC is clocked from one of the two options for operating in System Deep Sleep mode. Medium Frequency Clock is derived from System Deep Sleep operable Internal Main Oscillator (IMO) which consumes slightly more current than the LPOSC with the identical accuracy. LPOSC can be duty-cycled as explained in [LPOSC duty-cycling](#) section below. Therefore, use the LPOSC for additional power savings.

5.7.4.7 LPOSC duty-cycling

LPOSC provides an option to duty-cycle along with SAR ADC power, thus, providing power savings while SAR is powered-down in System Deep Sleep mode. If SAR ADC is configured to use Medium Frequency Clock, turn OFF the LPOSC to save power. When duty-cycling is enabled for LPOSC, ensure the Timer (which is the only trigger source in System Deep Sleep mode) is clocked using ILO or WCO.

5.7.4.8 Power-up delay

As described in [SAR ADCs](#) section, power-up delay allows the SAR ADC (mainly, the reference buffer), Analog Reference, LPOSC and Opamp (if used with duty-cycling) to settle before sampling and conversion process begins. Power-up delay is an 8-bit configuration value, clocked from either LPOSC or Medium Frequency Clock selected for the SAR ADC; i.e., Power-up delay can range from 0 to 127.5 µs.

The minimum value of power-up delay required is 20 µs. If your design includes external elements that require more settling time, increase the delay until there is no transient in the output. If the maximum delay is reached and output still has transient, modify your design to reduce the settling time required or throw away the initial few samples.

5.7.4.9 FIFO chain

Each SAR ADC in the device is equipped with its own FIFO of 64 samples depth to collect the measurement results in System Deep Sleep mode. FIFO chaining is possible wherein, FIFO associated with the second SAR ADC (SAR ADC 1) is chained to the FIFO of the first SAR ADC (SAR ADC 0), thereby, doubling the depth of the resulting FIFO. Note that this combined FIFO will load results only from SAR ADC 0. It helps to load larger number of results, while being in System Deep Sleep mode; thus, saving the power. Use FIFO chaining whenever faster sampling is used or when CPU processing is performed on large samples at a slower rate.

5.7.4.10 DMA

When the device wakes up, there are two options to read the FIFO data: using CPU or DMA. Using DMA saves the power and is useful when larger samples are required to be accumulated which cannot fit even in the chained FIFO. In this case, DMA can be used to move the FIFO data into memory when the CPU is in Sleep mode. When the required number of samples are accumulated, CPU can then be woken up for processing.

5 PSoC™ 6 application notes**5.7.5 Summary**

This application note provides an overview of the PSoC™ 62 MCU CY8C62x4, its low-power features and suggestions for implementing power-efficient applications.

~~DRAFT~~

5 PSoC™ 6 application notes

References

- 7

Application Notes

- [1] [AN228571](#) - Getting Started with PSoC™ 6 MCU on ModusToolbox™
- [2] [AN219528](#) - PSoC™ 6 MCU Low-Power Modes and Power Reduction Techniques

Code Examples

- [1] [CE230699](#) - PSoC™ 6 MCU SAR ADC Low-Power Sensing – Thermistor and Ambient Light Sensor
- [2] [CE230700](#) - PSoC™ 6 MCU Low-Power Analog Front End
- [3] [CE230701](#) - PSoC™ 6 MCU Simultaneous Sampling SAR ADCs

~~5 PSoC™ 6 application notes~~

~~5.7.6 Revision history~~

Document version	Date of release	Description of changes
**	2021-07-02	Initial release.
*A	2022-07-21	Template update.

5.8 AN85951 PSoC™ 4 and PSoC™ 6 MCU CAPSENSE™ design guide

About this document

-
- 8

Scope and purpose

The CAPSENSE™ design guide explains how to design capacitive touch sensing applications with the CAPSENSE™ feature in PSoC™ 4 and PSoC™ 6 MCU device families. The CAPSENSE™ feature offers unprecedented signal-to-noise ratio (SNR), best-in-class liquid tolerance, and a wide variety of sensors such as buttons, sliders, touchpads, and proximity sensors. This design guide explains the CAPSENSE™ operation, CAPSENSE™ design tools, performance tuning of the PSoC™ Creator and ModusToolbox™ CAPSENSE™ component and design considerations. This guide also introduces Fifth Generation CAPSENSE™ technology which has several advantages over the previous generation devices.

Different device families are available with CAPSENSE™ feature. If you have not chosen a particular device, or are new to capacitive sensing, see the [Getting started with CAPSENSE™ design guide](#). It helps you understand the advantages of CAPSENSE™ over mechanical buttons, CAPSENSE™ technology fundamentals, and to select the right device for your application. It also directs you to the right documentation, kits, or tools to help with your design.

Intended audience

This document is primarily intended for engineers who need to become familiar with the CAPSENSE™ design principles of PSoC™ 4 and PSoC™ 6 MCU devices.

5 PSoC™ 6 application notes

5.8.1 Introduction

5.8.1.1 Overview

Capacitive touch sensors are user interface devices that use human body capacitance to detect the presence of a finger on or near a sensor. CAPSENSE™ solutions bring elegant, reliable, and easy-to-use capacitive touch sensing functionality to your product.

This design guide focuses on the CAPSENSE™ feature in the PSoC™ 4 and PSoC™ 6 MCU families of devices. These are true programmable embedded system-on-chip, integrating configurable analog and digital peripheral functions, memory, radio, and a microcontroller on a single chip. These devices are highly flexible and can implement many functions such as ADC, DAC, and Bluetooth® LE in addition to CAPSENSE™, which accelerates time-to-market, integrates critical system functions, and reduces overall system cost.

This guide assumes that you are familiar with developing applications for PSoC™ 4 and PSoC™ 6 MCU using the PSoC™ Creator integrated design environment (IDE). If you are new to PSoC™ 4, see [AN79953 - Getting started with PSoC™ 4](#) or [AN92167 - Getting started with PSoC™ 4 Bluetooth® LE](#). If you are new to PSoC™ 6 MCU, see [AN221774 – Getting started with PSoC™ 6 MCU](#) and [AN210781 - Getting started with PSoC™ 6 MCU with Bluetooth® LE connectivity](#). If you are new to PSoC™ Creator, see the [PSoC™ Creator home page](#).

If you are new to ModusToolbox™, see [ModusToolbox™ IDE quick start guide](#).

This design guide helps you understand:

- CAPSENSE™ technology in PSoC™ 4 and PSoC™ 6 MCU
- Design and development tools available for [PSoC™ 4](#) and [PSoC™ 6 MCU](#) [PSoC™ 6 MCU CAPSENSE™](#)
- PSoC™ 6 MCU CAPSENSE™ PCB layout guidelines for PSoC 4 and PSoC 6 MCU
- Performance tuning of PSoC™ 4 and [PSoC™ 6 MCU CAPSENSE™ component](#)
- Applications using CAPSENSE™ Plus features such as motor control systems and induction cookers

5.8.1.2 CAPSENSE™ features

CAPSENSE™ in PSoC™ 4 and PSoC™ 6 MCU has the following features:

- Supports self-capacitance (CSD) and mutual-capacitance (CSX) based touch sensing on all CAPSENSE™-capable GPIO pins^{[12](#)}
- Provides the best in Class SNR allowing high sensitivity that provides high range proximity sensing (up to a 30-cm proximity-sensing distance) and liquid-tolerant operation (see [Liquid tolerance](#))
- High-performance sensing across a variety of overlay materials and varied thickness (see [CAPSENSE™ fundamentals](#), [Overlay material](#), and [Overlay thickness](#))
- [SmartSense](#) auto-tuning technology
- Pseudo random sequence (PRS) clock source, supports spread spectrum and programmable resistance switches for lower electromagnetic interference (EMI)
- Low power consumption with as low as 1.71 V operation and as low as 150 nA current consumption in hibernate mode

The PSoC™ 4100S Max device introduces Fifth-Generation CAPSENSE™ technology ([Ratiometric sensing technology](#)) and has the following additional features when compared to older generations.

- **Improved SNR:** Fifth-Generation CAPSENSE™ technology ([Ratiometric sensing technology](#)) significantly improves noise performance compared to previous generation devices
- **Improved refresh rate:** The better sensitivity of multi sense converter (MSC) requires less time to get similar signal as in previous generation therefore is able to achieve higher refresh rate. The two

¹² To achieve the best CAPSENSE™ performance, follow the recommendations in [Sensor pin selection](#) section

~~DRAFT~~ 5 PSoC™ 6 application notes

- independent MSC blocks which can scan the sensors in parallel improve the refresh rate further especially in use case where large numbers of sensors to be scanned
- **Improved CPU bandwidth:** Scan supported in both CPU mode and DMA mode. CPU mode is conventional interrupt driven mode, while DMA mode is capable of autonomous scanning which reduces the CPU bandwidth requirement to 18% compared to previous generation
- **Improved noise immunity:** Rail to rail swing is used as sense voltage, this provides maximum sense voltage and provides better immunity. In Fifth-Generation CAPSENSE™ technology full wave differential sensing is used for self-capacitance sensing and this cancels out noise induced from external environment to the sensor routings. This sensing technology is also better immune to power supply (V_{dd}) noise

5.8.1.3 PSoC™ 4 and PSoC™ 6 MCU CAPSENSE™ Plus features

You can create PSoC™ 4 CAPSENSE™ Plus applications that feature capacitive touch sensing and additional system functionality. The key features of these devices, in addition to CAPSENSE™ are:

- Arm® Cortex® -M0/M0+ CPU with single cycle multiply delivering up to 43 DMIPS at 48 MHz
- 1.71 V – 5.5 V operation over –40 to 85°C ambient
- Up to 128 KB of flash (CM0+ has > 2X code density over 8-bit solutions)
- Up to 16 KB of SRAM
- Up to 94 programmable GPIOs
- Independent center-aligned PWMs with complementary dead-band programmable outputs, synchronized ADC operation (ability to trigger the ADC at a customer-specifiable time in the PWM cycle), and synchronous refresh (ability to synchronize PWM duty cycle changes across all PWMs to avoid anomalous waveforms)
- Comparator-based triggering of PWM Kill signals (to terminate motor-driving when an over-current condition is detected)
- 12-bit 1 Msps ADC including sample-and-hold (S&H) capability with zero-overhead sequencing allowing the entire ADC bandwidth to be used for signal conversion and none used for sequencer overhead
- Opamps with comparator mode and SAR input buffering capability
- Segment LCD direct drive that supports up to four commons
- SPI/UART/I2C serial communication channels
- Bluetooth® LE communication compliant with version 4.0 and multiple features of version 4.1
- Programmable logic blocks, each having eight macrocells and a cascadable data path, called universal digital blocks (UDBs) for efficient implementation of programmable peripherals (such as I2S)
- Controller area network (CAN)
- Fully-supported PSoC™ Creator design entry, development, and debug environment providing:
 - Design entry and build (comprehending analog routing)
 - Components for all fixed-function peripherals and common programmable peripherals
 - Documentation and training modules
- Support for porting builds to MDK Arm® environment (previously known as RealView) and others
- Support for Eclipse integrated development environment (IDE) for ModusToolbox™

The main features of PSoC™ 6 MCU device, in addition to CAPSENSE™ are:

- Single CPU devices (Arm® Cortex® -M4), dual CPU devices (Arm® Cortex® -M4 and Cortex® -M0+). Support for inter-processor communication in hardware
- 1.71 V - 3.6 V device operating voltage with user selectable core logic operation at either 1.1 V or 0.9 V
- Up to 2 MB of flash memory and up to 1 MB of SRAM
- Up to 78 GPIOs that can be used for analog, digital, CAPSENSE™, or segment LCD functions

5 PSoC™ 6 application notes

- ~~DRAFT~~
- Programmable analog blocks: Two opamps, configurable PGAs, comparators, 12-bit 1 Msps SAR ADC, 12-bit voltage mode DAC
 - Programmable digital blocks, communication interfaces
 - 12 UDBs, 32 TCPWMs configurable as 16-bit/32-bit timer, counter, PWM, or quadrature decoder
 - Up to 13 serial communication block (SCB) configurable as I2C, SPI, or UART interfaces. See the [Device datasheet](#) for more details
 - Audio subsystem with one I2S interface and two PDM channels
 - SMIF interface with support for execute-in-place from external quad SPI flash memory and on-the-fly encryption and decryption
 - Bluetooth® Smart connectivity with Bluetooth® LE 5.0 (applicable only to PSoC™ 6 MCU with Bluetooth® LE family of devices)

See [AN64846 - Getting started with CAPSENSE™](#) to select an appropriate CAPSENSE™ device based on your requirements.

5.8.1.4 CAPSENSE™ design flow

[Figure 191](#) illustrates the product design cycle with capacitive sensing; the information in this guide is highlighted in green. provides links to the supporting documents for each of the numbered tasks in [Figure 191](#).

5 PSoC™ 6 application notes

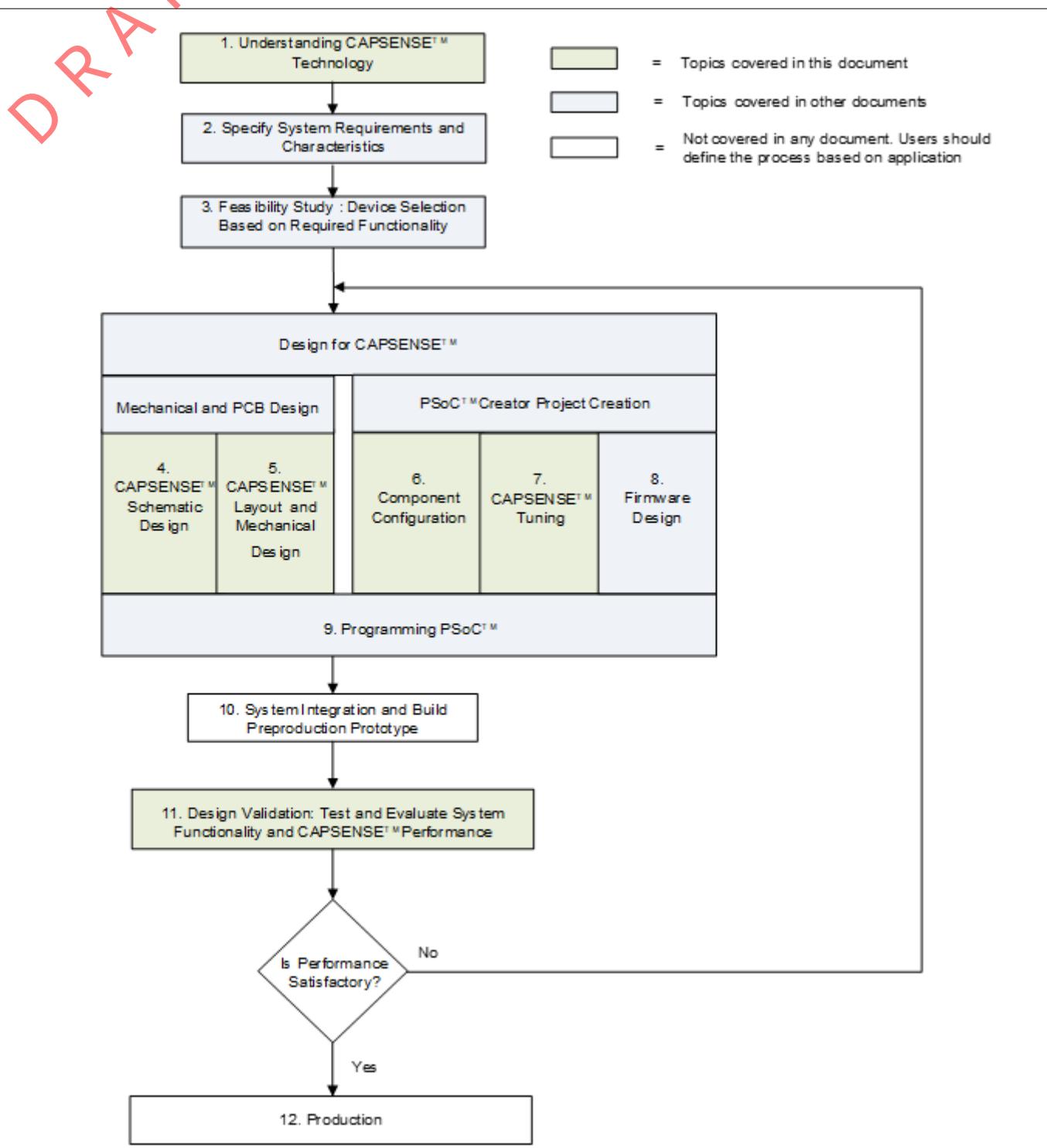


Figure 191 CAPSENSE™ product design flow

Table 34 Supporting documentation

Steps in flowchart	Supporting documentation	
	Name	Chapter
1. Understanding CAPSENSE™	CAPSENSE™ design guide (This document) Getting started with CAPSENSE™	Chapter 5.8.2 and Chapter 5.8.3 –

(table continues...)

Reference manual

5 PSoC™ 6 application notes

Table 34 (continued) Supporting documentation

Steps in flowchart	Supporting documentation	Chapter
Name		
2. Specify requirements	Getting started with CAPSENSE™	–
3. Feasibility study	PSoC™ 4 datasheet PSoC™ 4 Bluetooth® LE datasheet PSoC™ 6 MCU datasheet	–
	AN64846 – Getting started with CAPSENSE™ design guide AN79953 – Getting started with PSoC™ 4 AN91267 – Getting started with PSoC™ 4 Bluetooth® LE AN221774 – Getting started with PSoC™ 6 MCU	–
4. Schematic design	CAPSENSE™ design guide (This document)	Chapter 5.8.7
5. Layout design	CAPSENSE™ design guide (This document)	Chapter 5.8.7
6. Component configuration	PSoc™ CAPSENSE™ Component datasheet/middleware document CAPSENSE™ design guide (This document)	– Chapter 5.8.5
7. Performance tuning	PSoc™ CAPSENSE™ design guide (This document)	Chapter 5.8.5
8. Firmware design	PSoc™ Component datasheet/middleware document	–
	PSoc™ Creator Example projects Download ModusToolbox™ here . See the ModusToolbox™ related documents: ModusToolbox™ release notes ModusToolbox™ user guide ModusToolbox™ quick start guide ModusToolbox™ CAPSENSE™ configurator guide ModusToolbox™ CAPSENSE™ tuner guide PSoc™ Creator to ModusToolbox™ porting guide	–
9. Programming PSoC™	PSoc™ Creator user guide for in-IDE programming PSoC™ Programmer home page and MiniProg3 user guide for standalone programming	–
10. Prototype	–	–
11. Design validation	CAPSENSE™ design guide (This document)	Chapter 5.8.5
12. Production	–	–

5 PSoC™ 6 application notes

5.8.2 CAPSENSE™ technology

Capacitive touch sensing technology measures changes in capacitance between a plate (the sensor) and its environment to detect the presence of a finger on or near a touch surface.

5.8.2.1 CAPSENSE™ fundamentals

A typical CAPSENSE™ sensor consists of a copper pad of proper shape and size etched on the surface of a PCB. A non-conductive overlay serves as the touch surface for the button, as [Figure 192](#) shows.



Figure 192 Capacitive touch sensor

PCB traces and vias connect the sensor pads to PSoC™ GPIOs that are configured as CAPSENSE™ sensor pins. As [Figure 193](#) shows, the self-capacitance of each electrode is modeled as C_{SX} and the mutual capacitance between electrodes is modeled as C_{MX} . CAPSENSE™ circuitry internal to the PSoC™ converts these capacitance values into equivalent digital counts (see [Chapter 5.8.3](#) for details). These digital counts are then processed by the CPU to detect touches.

CAPSENSE™ also requires external capacitor C_{MOD} or C_{INT} for self-capacitance sensing and mutual-capacitance sensing. For third- and fourth-generation CAPSENSE™ architecture, a single C_{MOD} capacitor is required for self-capacitance sensing and C_{INTA} and C_{INTB} capacitors for mutual-capacitance sensing. If shield electrode is implemented for liquid tolerance, or for large proximity sensing distance, an additional C_{TANK} capacitor may be required. For Fifth-Generation CAPSENSE™ architecture, two C_{MOD} capacitors are required for both self-capacitance and mutual-capacitance sensing for each channel. These external capacitors are connected between a dedicated GPIO pin and ground. [Table 68](#) list the recommended values of the external capacitors.

5 PSoC™ 6 application notes

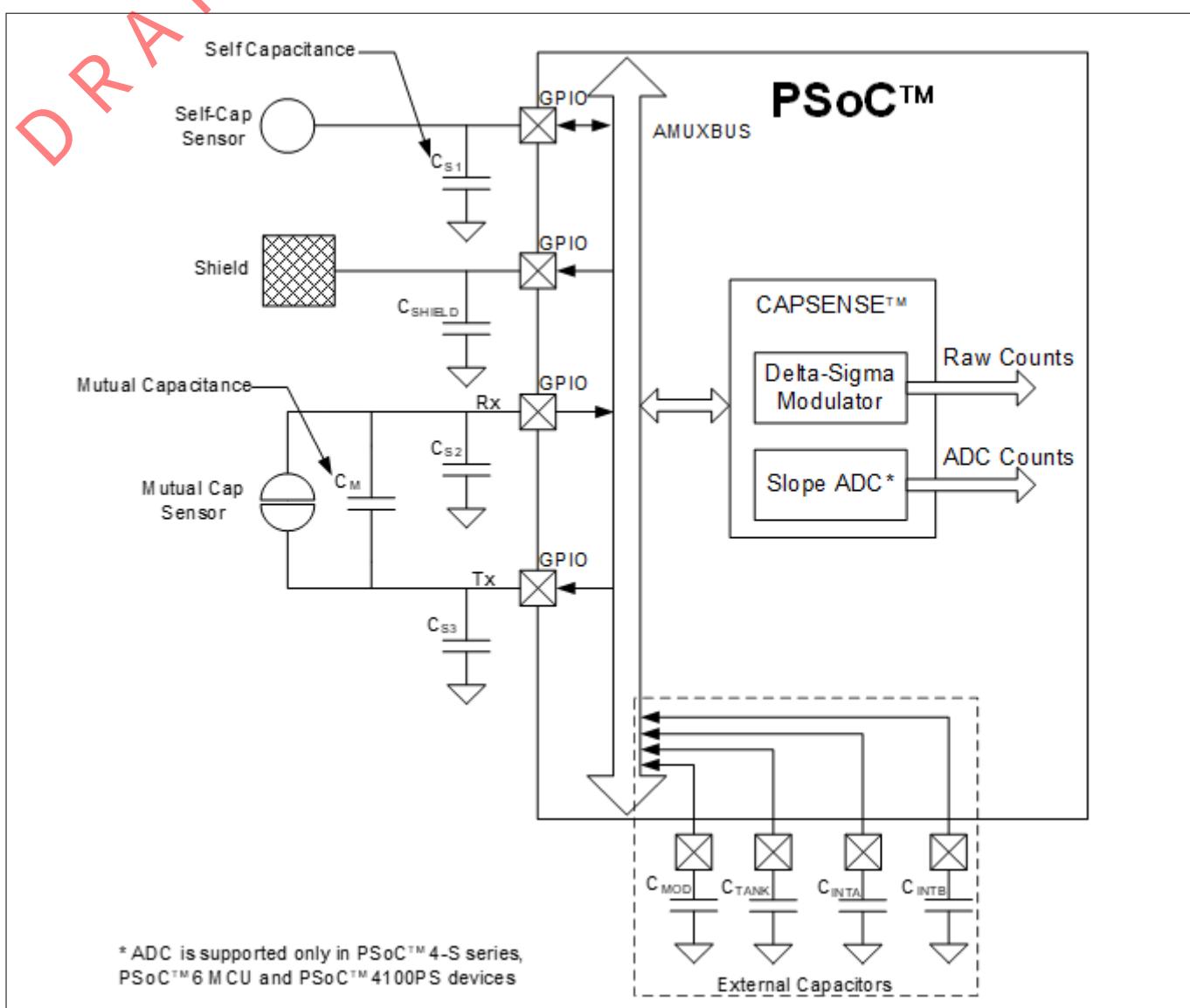


Figure 193 PSoC™ device, sensors, and external capacitors

The capacitance of the sensor in the absence of a touch is called the parasitic capacitance, C_p . C_p results from the electric field between the sensor (including the sensor pad, traces, and vias) and other conductors in the system such as the ground planes, traces, and any metal in the product's chassis or enclosure. The GPIO and internal capacitances of PSoC™ also contribute to the parasitic capacitance. However, these internal capacitances are typically very small compared to the sensor capacitance.

5.8.2.1.1 Self-capacitance sensing

Figure 194 shows how a GPIO pin is connected to a sensor pad by traces and vias for self-capacitance sensing. Typically, a ground (GND) hatch surrounds the sensor pad to isolate it from other sensors and traces. Although Figure 194 shows some field lines around the sensor pad, the actual electric field distribution is very complex.

5 PSoC™ 6 application notes

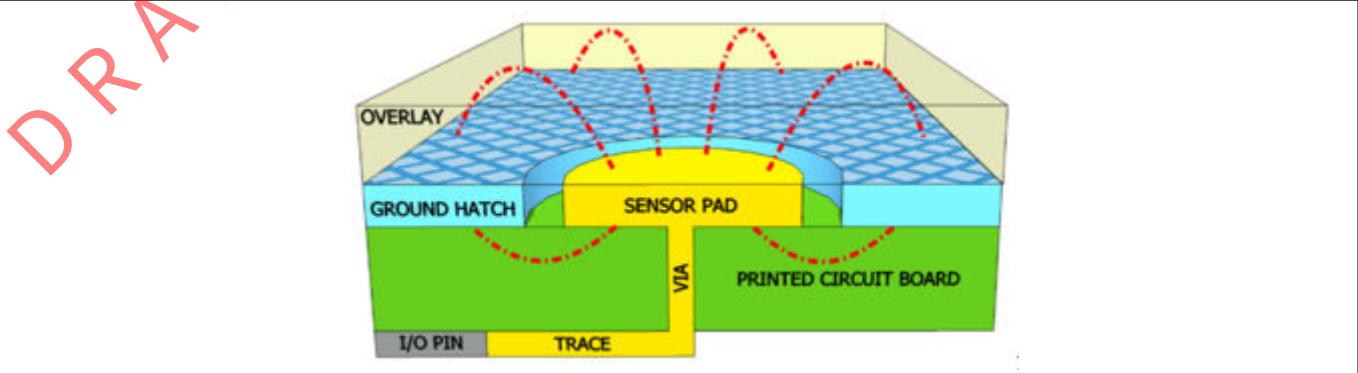


Figure 194 Parasitic capacitance

When a finger is present on the overlay, the conductive nature and large mass of the human body forms a grounded, conductive plane parallel to the sensor pad, as [Figure 195](#) shows.

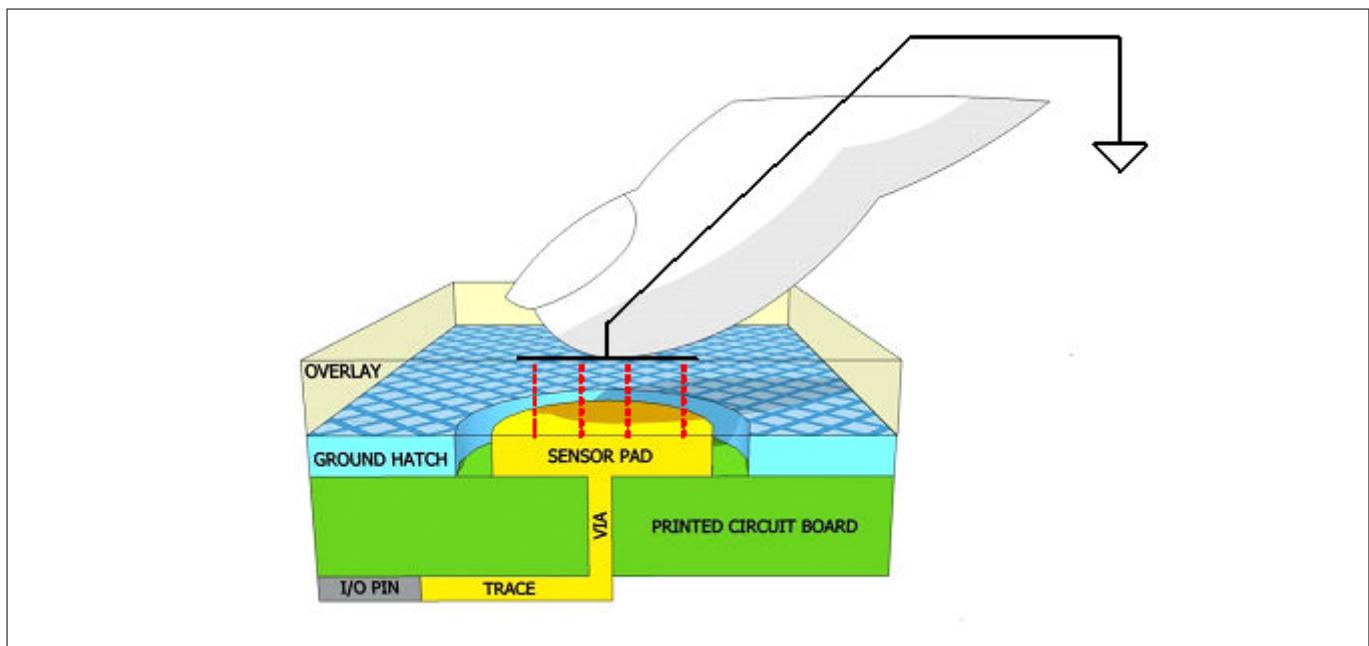


Figure 195 Finger capacitance

This arrangement forms a parallel plate capacitor. The capacitance between the sensor pad and the finger is shown in [Equation 1](#).

$$C_F = \frac{\epsilon_0 \epsilon_r A}{d}$$

Equation 2 Finger capacitance

Where:

ϵ_0 = Free space permittivity

ϵ_r = Relative permittivity of overlay

A = Area of finger and sensor pad overlap

d = Thickness of the overlay

C_F = Finger capacitance.

5 PSoC™ 6 application notes

~~PAGE~~
~~REDACTED~~

C_P and C_F are parallel to each other because both represent the capacitance between the sensor pin and ground. Therefore, the total capacitance C_S of the sensor, when the finger is present on the sensor, is the sum of C_P and C_F .

$$C_S = C_P + C_F$$

Equation 3 Total sense capacitance when finger is present on sensor

In the absence of touch, C_S is equal to C_P .

PSoC™ converts the capacitance C_S into equivalent digital counts called raw counts. Because a finger touch increases the total capacitance of the sensor pin, an increase in the raw counts indicates a finger touch. Refer to the CSD specification in [Device datasheet/Component datasheet/middleware document](#) document to learn about the supported CP range for a given device with which the recommended SNR can be achieved.

5.8.2.1.2 Mutual-capacitance sensing

[Figure 196](#) shows the button sensor layout for mutual-capacitance sensing. Mutual-capacitance sensing measures the capacitance between two electrodes, transmit (Tx) electrode and receive (Rx) electrode.

In a mutual-capacitance sensing system, a digital voltage signal switching between VDDIO¹³⁾ or VDDD¹⁴⁾ (if VDDIO is not supported by the device) and GND is applied to the Tx pin and the amount of charge received on the Rx pin is measured. The amount of charge received on the Rx electrode is directly proportional to the mutual-capacitance (C_M) between the two electrodes.

When a finger is placed between the Tx and Rx electrodes, the mutual-capacitance decreases to C_M^1 , as shown in [Figure 197](#). Because of the reduction in the mutual-capacitance, the charge received on the Rx electrode also decreases. The CAPSENSE™ system measures the amount of charge received on the Rx electrode to detect a touch/no touch condition.

¹³

¹⁴ VDDD is the device power supply for digital section.

5 PSoC™ 6 application notes

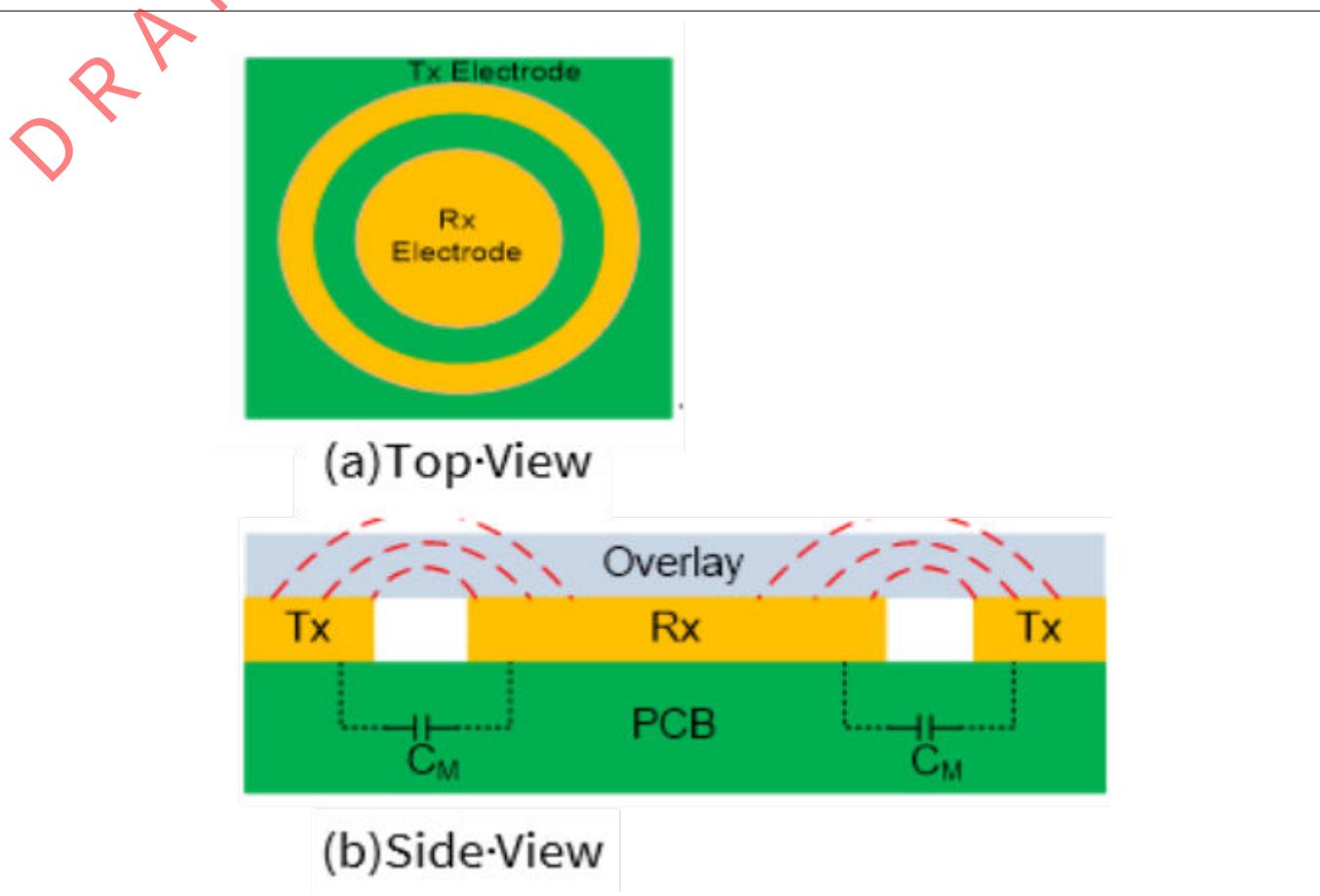


Figure 196 Mutual-capacitance sensing

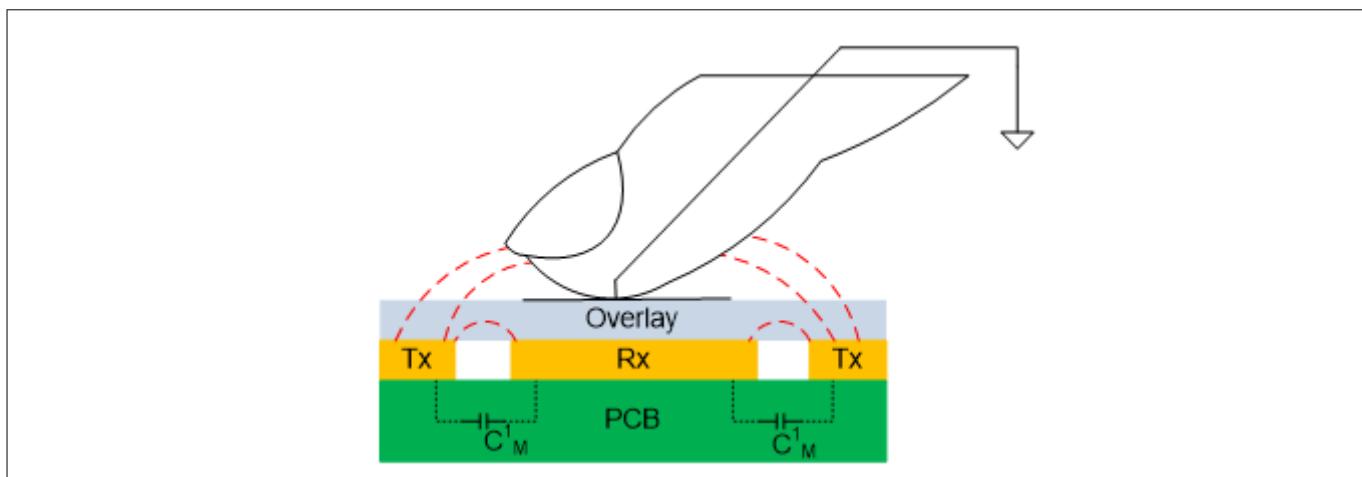


Figure 197 Mutual-capacitance with finger touch

5.8.2.2 Capacitive touch sensing method

PSoC™ uses patented capacitive touch-sensing method CAPSENSE™ sigma delta (CSD) for self-capacitance sensing and CAPSENSE™ crosspoint (CSX) for mutual-capacitance scanning. The CSD and CSX touch sensing methods provide the industry's best-in-class [Signal-to-noise ratio \(SNR\)](#). These sensing methods are a combination of hardware and firmware techniques.

~~5 PSoC™ 6 application notes~~

~~5.8.2.2.1 CAPSENSE™ sigma delta (CSD)~~

~~Figure 198~~ shows a simplified block diagram of the CSD method.

In CSD, each GPIO has a switched-capacitance circuit that converts C_S into an equivalent current. An analog MUX (AMUX) selects one of the sensor currents and feeds it into the current to digital converter. The current to digital converter is similar to a delta sigma ADC. The output count of the current to digital converter, known as raw count, is a digital value that is proportional to the self-capacitance between the electrodes.

$$\text{raw count} = G_{\text{CSD}} C_S$$

Equation 4 Raw count and sensor capacitance relationship in CSD

Where,

G_{CSD} = Capacitance to digital conversion gain of CSD

C_S = Self-capacitance of the electrode

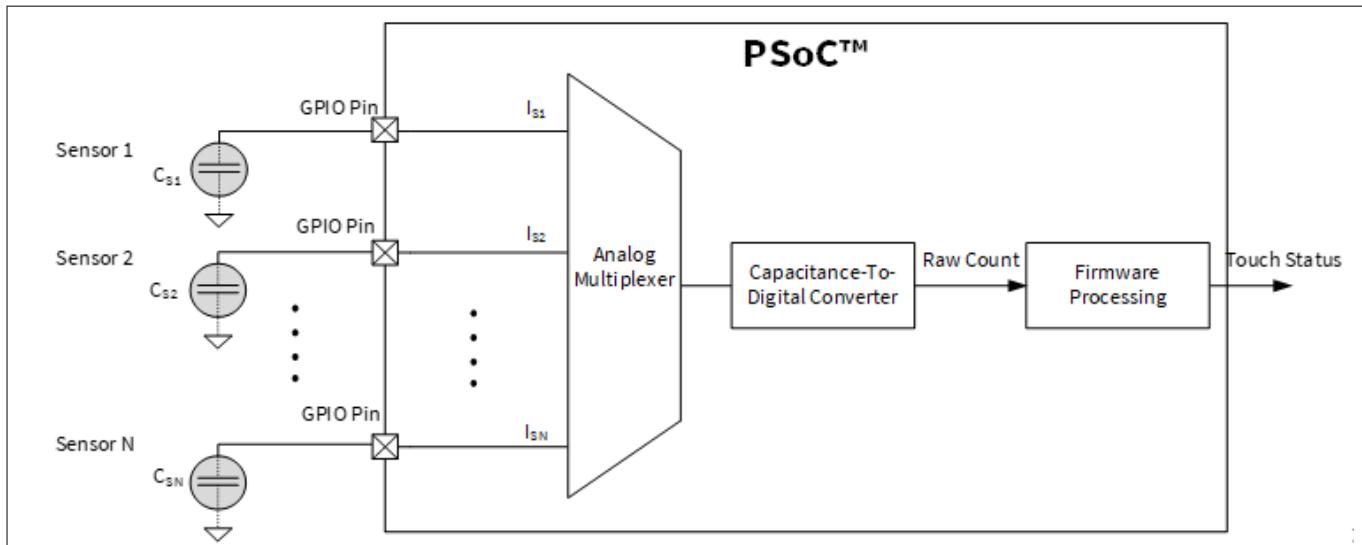


Figure 198 Simplified diagram of CSD method

Figure 200 illustrates a plot of raw count over time. When a finger touches the sensor, the C_S increases from C_P to $C_P + C_F$, and the raw count increases. By comparing the change in raw count to a predetermined threshold, logic in firmware decides whether the sensor is active (finger is present).

5.8.2.2.2 CAPSENSE™ crosspoint (CSX)

Figure 199 shows the simplified block diagram of the CSX method.

5 PSoC™ 6 application notes

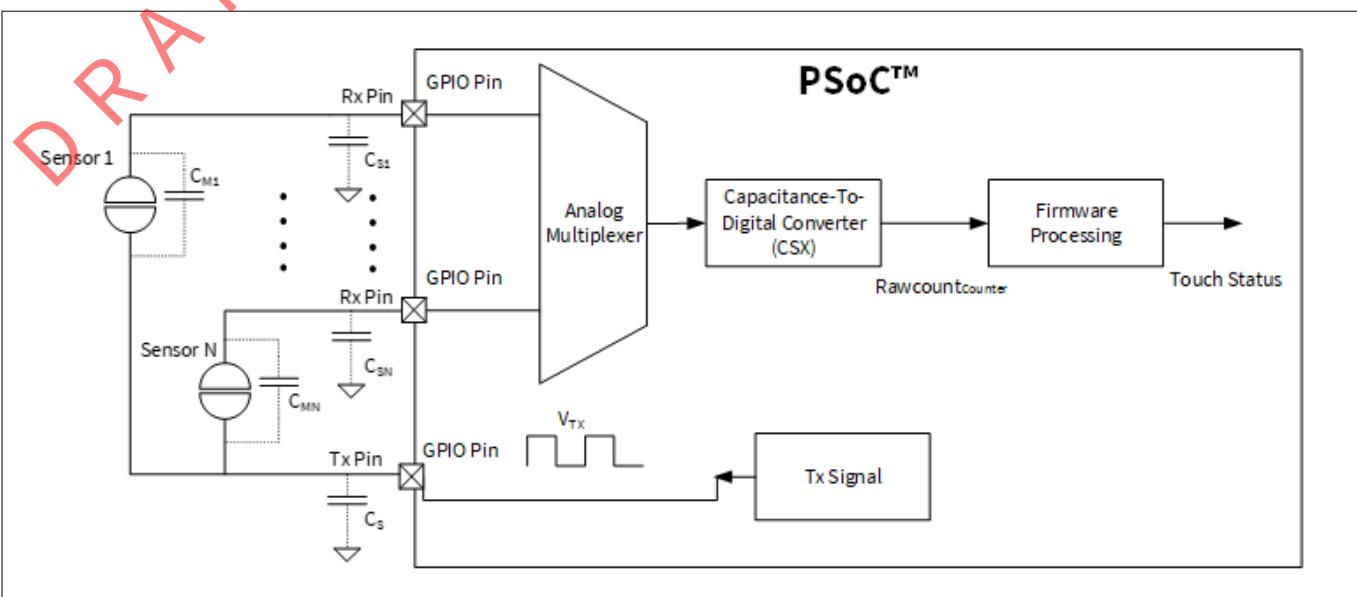


Figure 199 Simplified diagram of CAPSENSE™ crosspoint (CSX) method

With CSX, a voltage on the Tx electrode couples charge on to the RX electrode. This charge is proportional to the mutual capacitance between the Tx and Rx electrodes. An analog MUX then selects one of the Rx electrodes and feeds it into the current to digital converter.

The output count of the current to digital converter, $\text{Rawcount}_{\text{Counter}}$, is a digital value that is proportional to the mutual-capacitance between the Rx and Tx electrodes as shown in [Equation 5](#).

$$\text{Rawcount}_{\text{Counter}} = G_{\text{CSX}} C_M$$

Equation 5 Raw count and sensor capacitance relationship in CSX

Where,

G_{CSX} = Capacitance to digital conversion gain of mutual capacitance method

C_M = Mutual-capacitance between two electrodes

[Figure 200](#) illustrates a plot of raw count over time. When a finger touches the sensor, C_M decreases from C_M to $C_{M'}^1$ (see [Figure 197](#)) hence the counter output decreases. The firmware normalizes the raw count such that the raw counts go high when C_M decreases. This maintains the same visual representation of raw count between CSD and CSX methods. By comparing the change in raw count to a predetermined threshold, logic in firmware decides whether the sensor is active (finger is present). The normalized inverted raw count is computed using [Equation 16](#).

5 PSoC™ 6 application notes

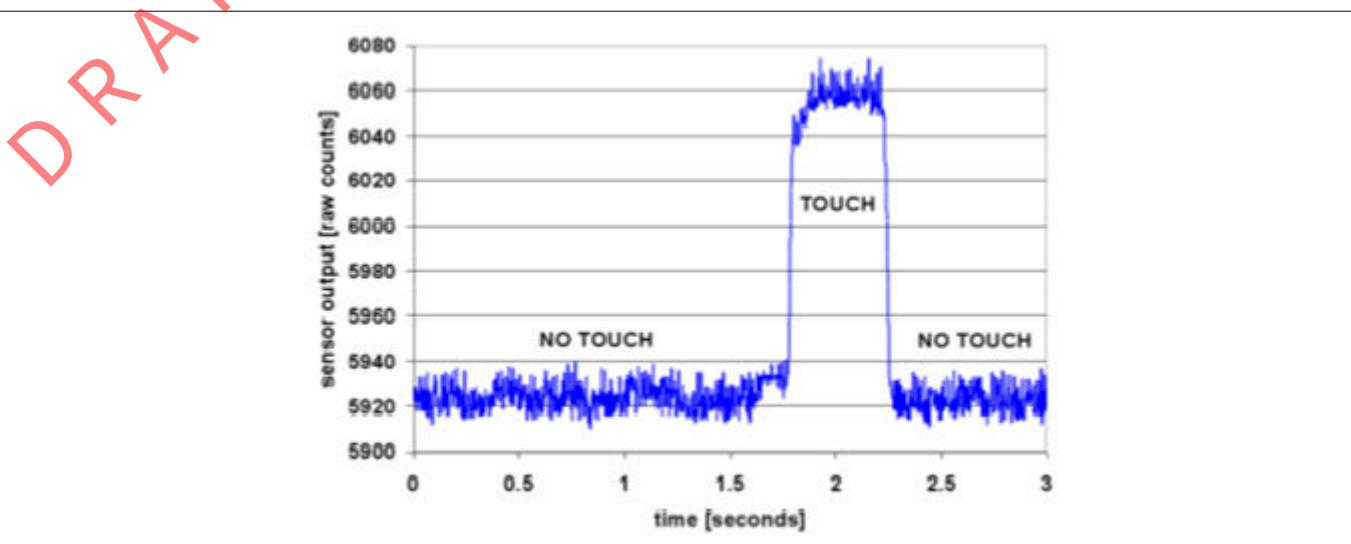


Figure 200 Raw count versus time

For an in-depth discussion of the PSoC™ 4 and PSoC™ 6 CAPSENSE™ CSD and CSX blocks, see chapter [PSoC™ 4](#) and [PSoC™ 6 MCU CAPSENSE™](#).

5.8.2.3 Signal-to-noise ratio (SNR)

In practice, the raw counts vary due to inherent noise in the system. CAPSENSE™ noise is the peak-to-peak variation in raw counts in the absence of a touch, as [Figure 201](#) shows.

A well-tuned CAPSENSE™ system reliably discriminates between the ON and OFF states of the sensors. To achieve good performance, the CAPSENSE™ signal must be significantly larger than the CAPSENSE™ noise. SNR is defined as the ratio of CAPSENSE™ signal to CAPSENSE™ noise. SNR is the most important performance parameter of a CAPSENSE™ sensor.

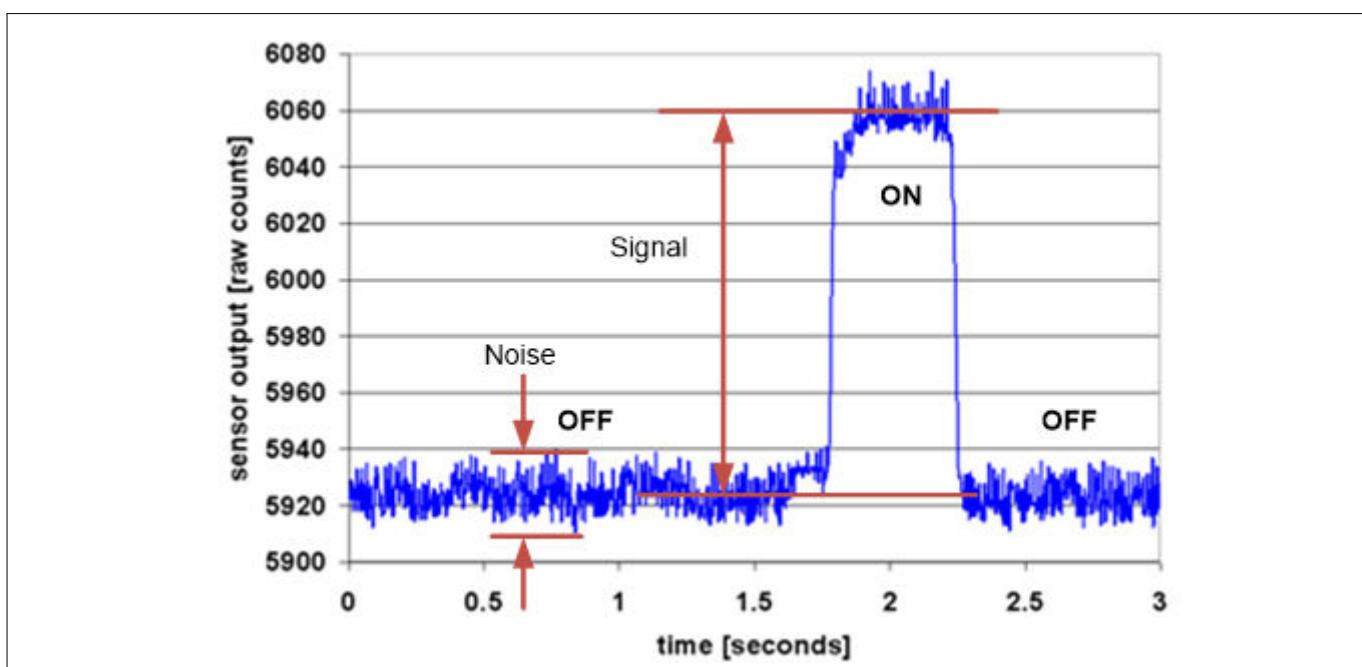


Figure 201 SNR

In this example, the average level of raw count in the absence of a touch is 5925 counts. When a finger is placed on the sensor, the average raw count increases to 6060 counts, which means the signal is $6060 - 5925 = 135$

~~5 PSoC™ 6 application notes~~

counts. The minimum value of the raw count in the OFF state is 5912 and the maximum value is 5938 counts. Therefore, the CAPSENSE™ noise is $5938 - 5912 = 26$ counts. This results in an SNR of $135/26 = 5.2$.

The minimum SNR recommended for a CAPSENSE™ sensor is 5. This 5:1 ratio comes from best practice threshold settings, which enable enough margin between signal and noise in order to provide reliable ON/OFF operation.

5.8.2.4 CAPSENSE™ widgets

CAPSENSE™ widgets consist of one or more CAPSENSE™ sensors, which as a unit represent a certain type of user interface. CAPSENSE™ widgets are broadly classified into four categories – buttons (zero-dimensional), sliders (one-dimensional), touchpads/trackpads (two-dimensional), and proximity sensors (three-dimensional). [Figure 202](#) shows button, slider, and proximity sensor widgets. This section explains the basic concepts of different CAPSENSE™ widgets. For a detailed explanation of sensor construction, see [Sensor construction](#).

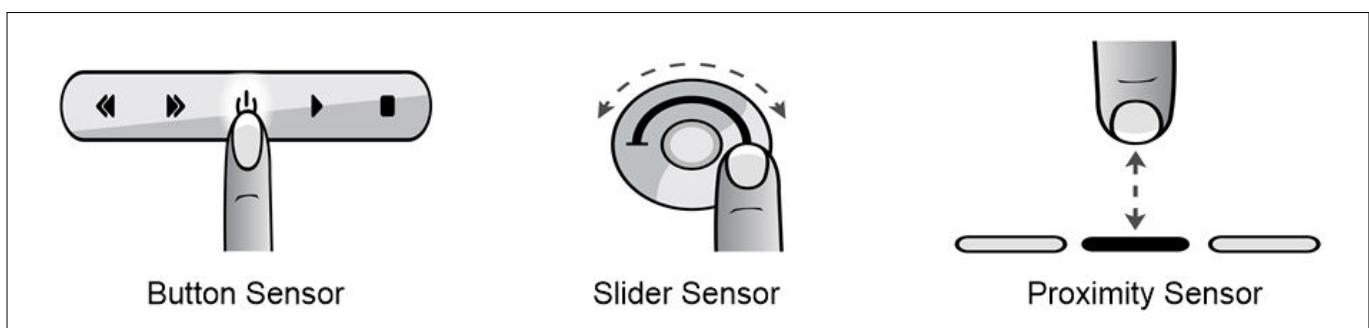


Figure 202 **Several types of widgets**

5.8.2.4.1 Buttons (zero-dimensional)

CAPSENSE™ buttons replace mechanical buttons in a wide variety of applications such as home appliances, medical devices, white goods, lighting controls, and many other products. It is the simplest type of CAPSENSE™ widget, consisting of a single sensor. A CAPSENSE™ button gives one of two possible output states: active (finger is present) or inactive (finger is not present). These two states are also called ON and OFF states, respectively.

For the self-capacitance (CSD) sensing method, a simple CAPSENSE™ button consists of a circular copper pad connected to a PSoC™ GPIO with a PCB trace. The CAPSENSE™ button is surrounded by grounded copper hatch that isolates it from other buttons and traces. A circular gap separates the button pad and the ground hatch. Each button requires one PSoC™ GPIO. These buttons can be constructed using any conductive material on a non-conductive substrate; for example, indium tin oxide on a glass substrate, or silver ink on a non-conductive film. Even metallic springs can be used as button sensors; see [Sensor construction](#) for more details.

5 PSoC™ 6 application notes

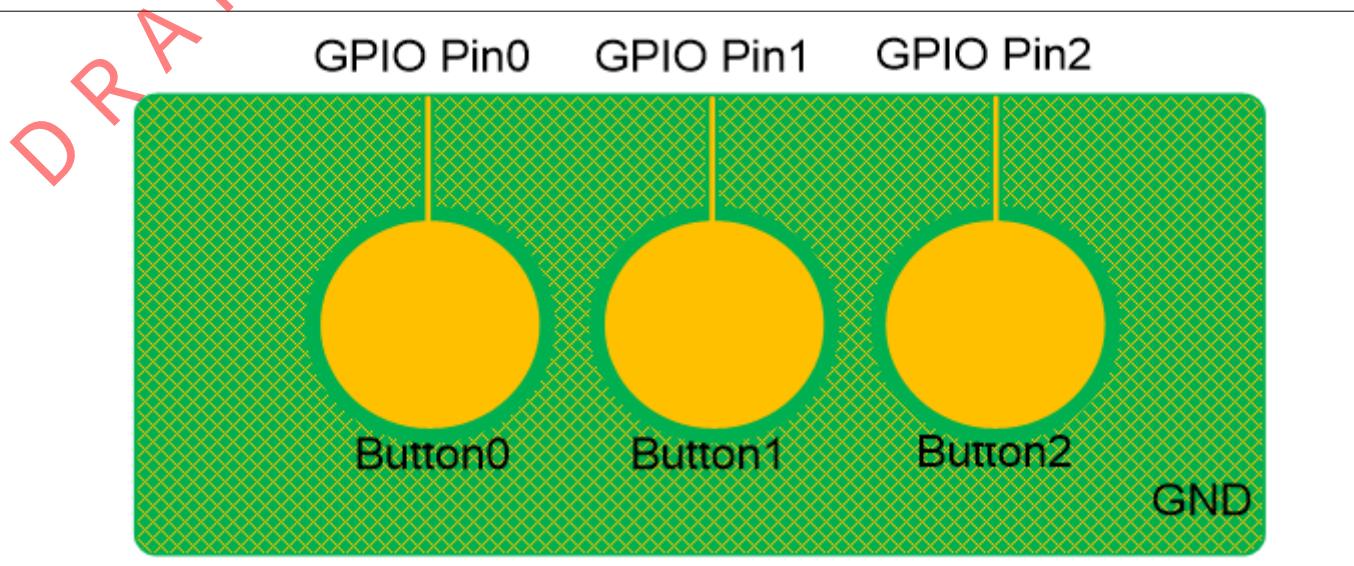


Figure 203 Simple CAPSENSE™ buttons

For the mutual-capacitance (CSX) sensing method, each button requires one GPIO pin configured as Tx electrode and one GPIO pin configured as Rx electrode. The Tx can be shared across multiple buttons, as shown in [Figure 204](#).

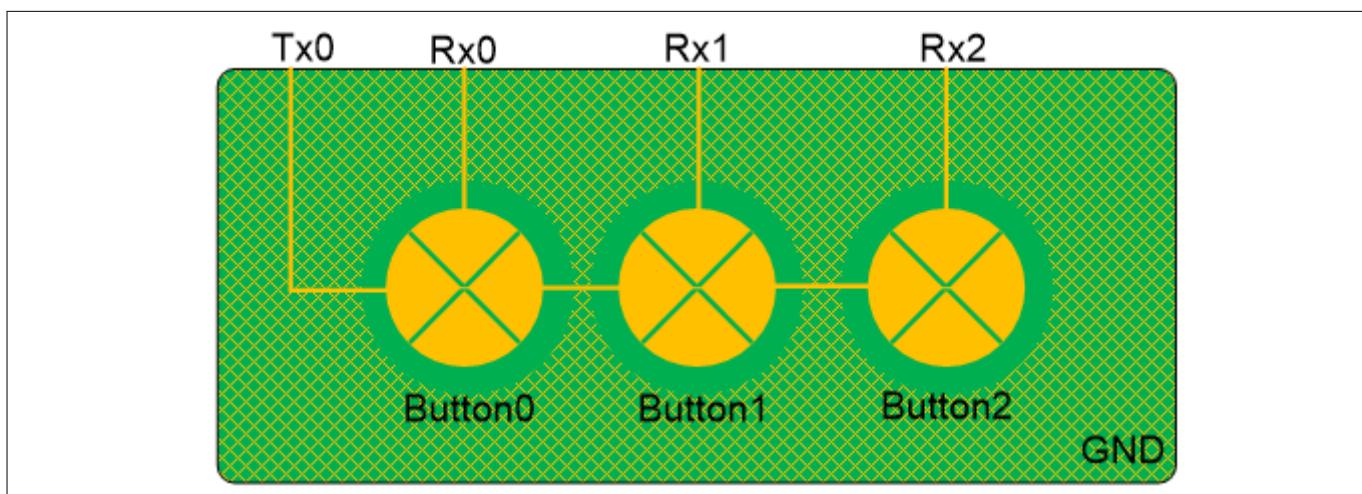
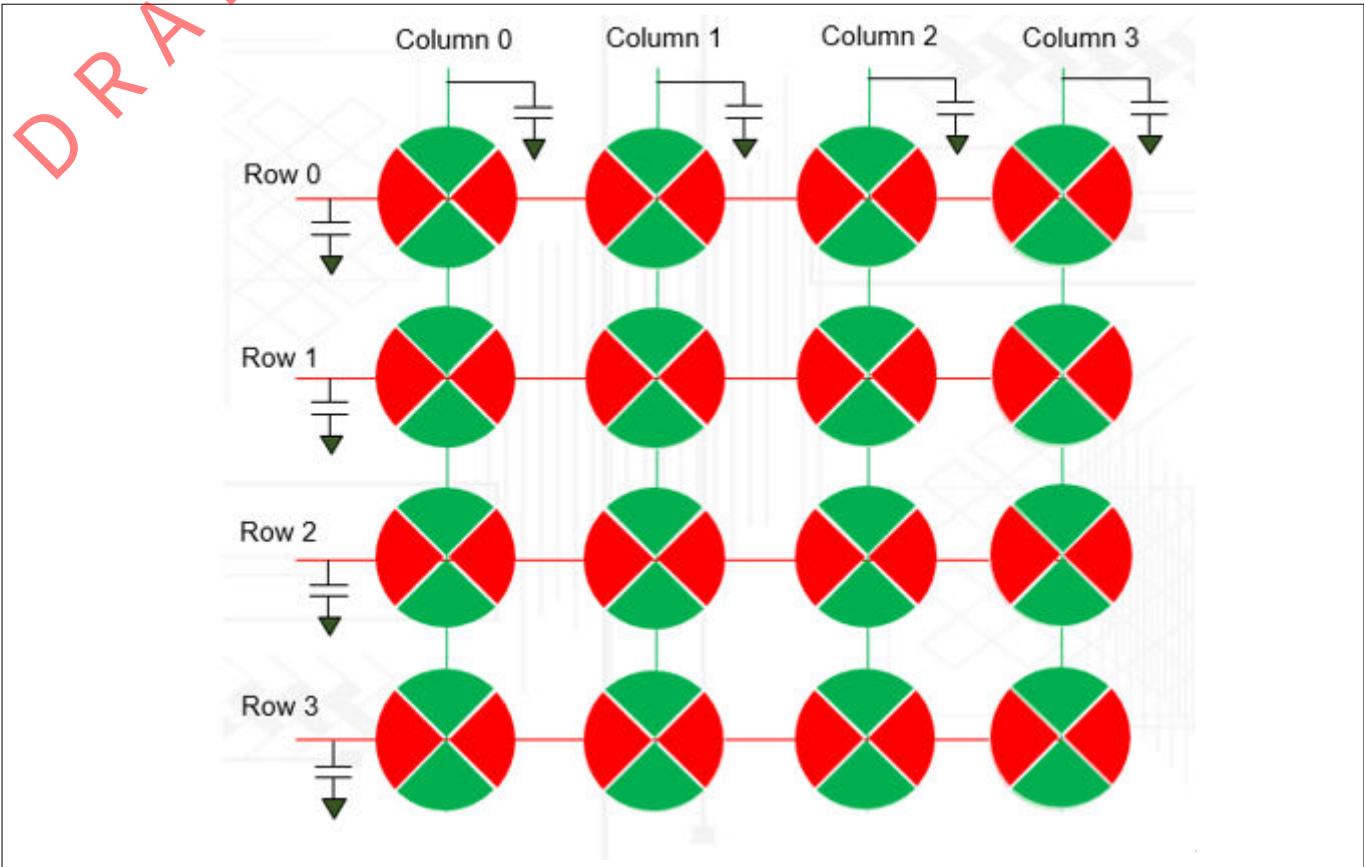


Figure 204 Simple CAPSENSE™ buttons for mutual-capacitance sensing method

If the application requires many buttons (for example in a calculator keypad or a QWERTY keyboard), you can arrange the CAPSENSE™ buttons in a matrix, as [Figure 205](#) shows. This allows a design to have multiple buttons per GPIO. For example, the 16-button design in [Figure 205](#) requires only eight GPIOs.

5 PSoC™ 6 application notes

**Figure 205 Matrix buttons based on CSD**

A matrix button design has two groups of capacitive sensors: row sensors and column sensors. The matrix button architecture can be used for both self-capacitance (CSD) and mutual-capacitance (CSX) methods.

In CSD mode, each button consists of a row sensor and a column sensor, as [Figure 205](#) shows. When a button is touched, both row and column sensors of that button become active. The CSD-based matrix button should be used only if the user is expected to touch one button at a time. If the user touches more than one diagonally opposite buttons, the finger location cannot be resolved as [Figure 206](#) shows. This effect is called as ghost effect, which is considered an invalid condition.

5 PSoC™ 6 application notes

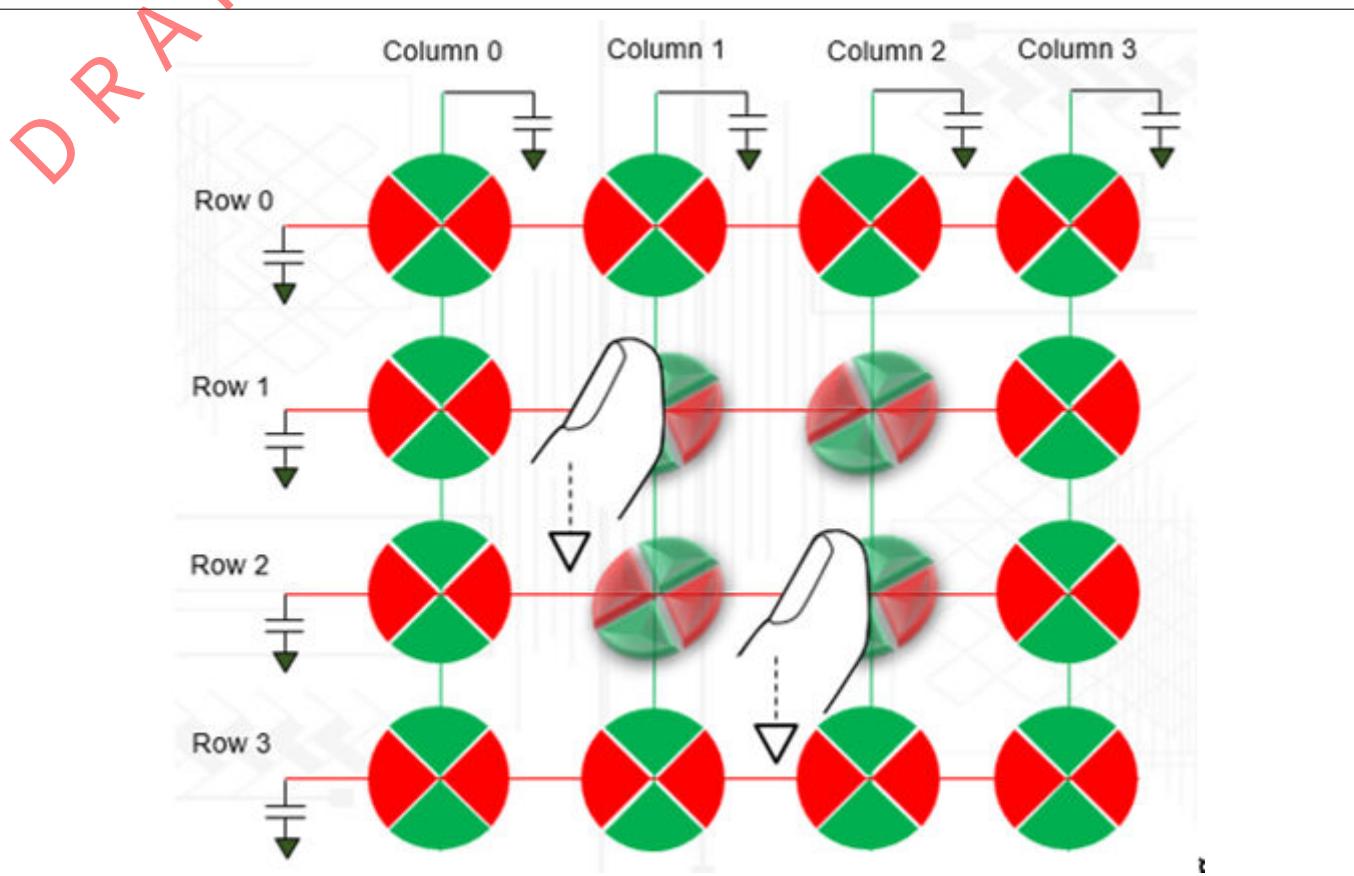


Figure 206 Ghost effect in matrix button based on CSD

Mutual-capacitance is the recommended sensing method for matrix buttons because this method is not affected from the ghost touch phenomena and provides better SNR for high C_p sensors. This is because it senses mutual-capacitance formed at each intersection rather than sensing rows and columns as shown in [Figure 207](#). Applications that require simultaneous sensing of multiple buttons, such as a keyboard with **Shift**, **Ctrl**, and **Alt** keys can use CSX sensing method or you should design the **Shift**, **Ctrl**, and **Alt** keys as individual CSD buttons.

5 PSoC™ 6 application notes

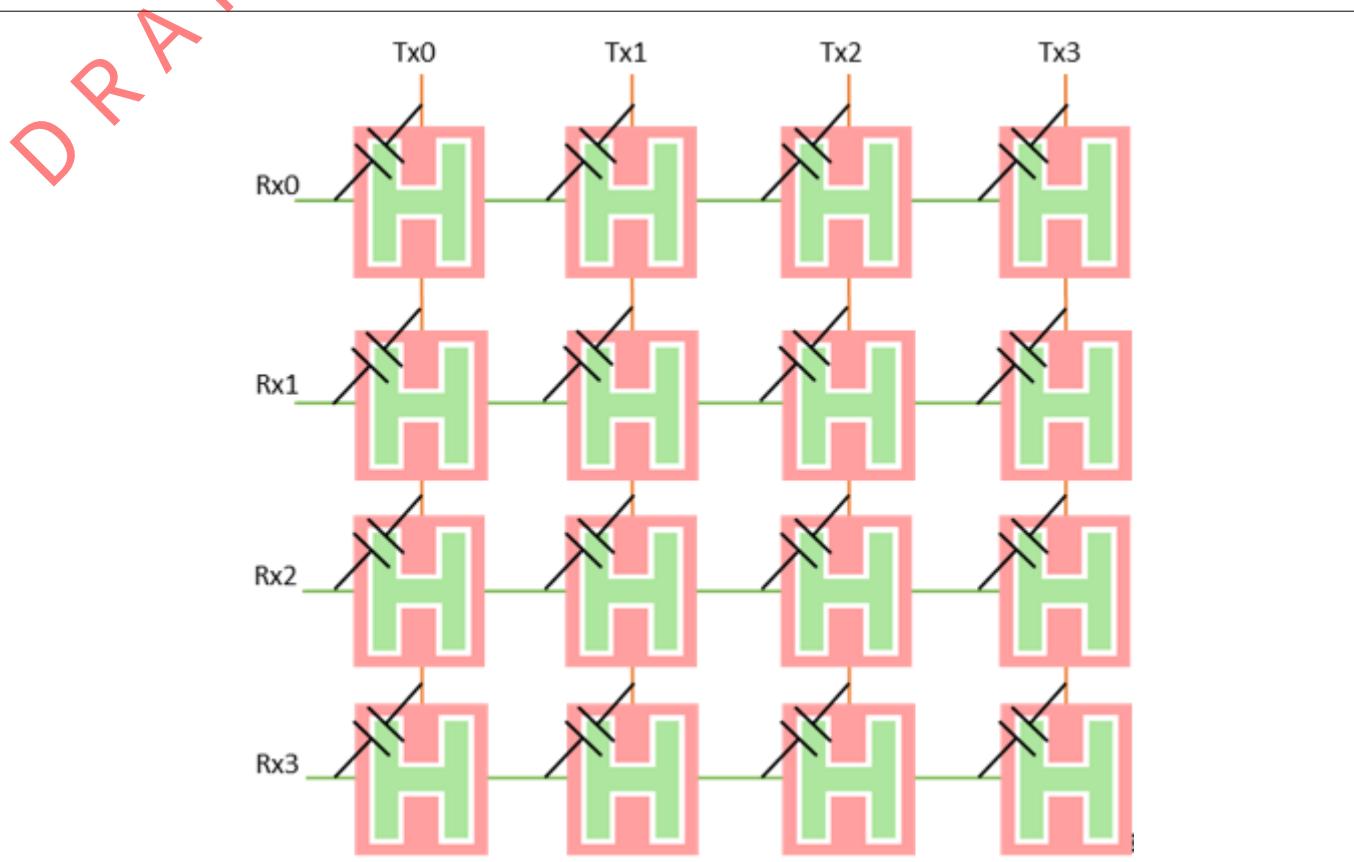


Figure 207 Matrix button based on CSX

Note: Scanning a matrix keypad using CSX sensing method may require a longer overall scan time than the CSD sensing method. This is because the CSD sensing method scans rows and columns as sensors, while the CSX sensing method scans each intersection as a sensor.

5.8.2.4.2 Sliders (one-dimensional)

Sliders are used when the required input is in the form of a gradual increment or decrement. Examples include lighting control (dimmer), volume control, graphic equalizer, and speed control. Currently, the CAPSENSE™ Component in PSoC™ Creator and ModusToolbox™ supports only self-capacitance-based sliders. Mutual capacitance-based sliders will be supported in future version of component.

A slider consists of a one-dimensional array of capacitive sensors called segments, which are placed adjacent to one another. Touching one segment also results in partial activation of adjacent segments. The firmware processes the raw counts from the touched segment and the nearby segments to calculate the position of the geometric center of the finger touch, which is known as the **centroid position**.

The actual resolution of the calculated centroid position is much higher than the number of segments in a slider. For example, a slider with five segments can resolve at least 100 physical finger positions. This high resolution gives smooth transitions of the centroid position as the finger glides across a slider.

In a linear slider, the segments are arranged inline, as Figure 208 shows. Each slider segment connects to a PSoC™ GPIO. A zigzag pattern (double chevron) is recommended for slider segments. This layout ensures that when a segment is touched, the adjacent segments are also partially touched, which aids estimation of the centroid position.

5 PSoC™ 6 application notes

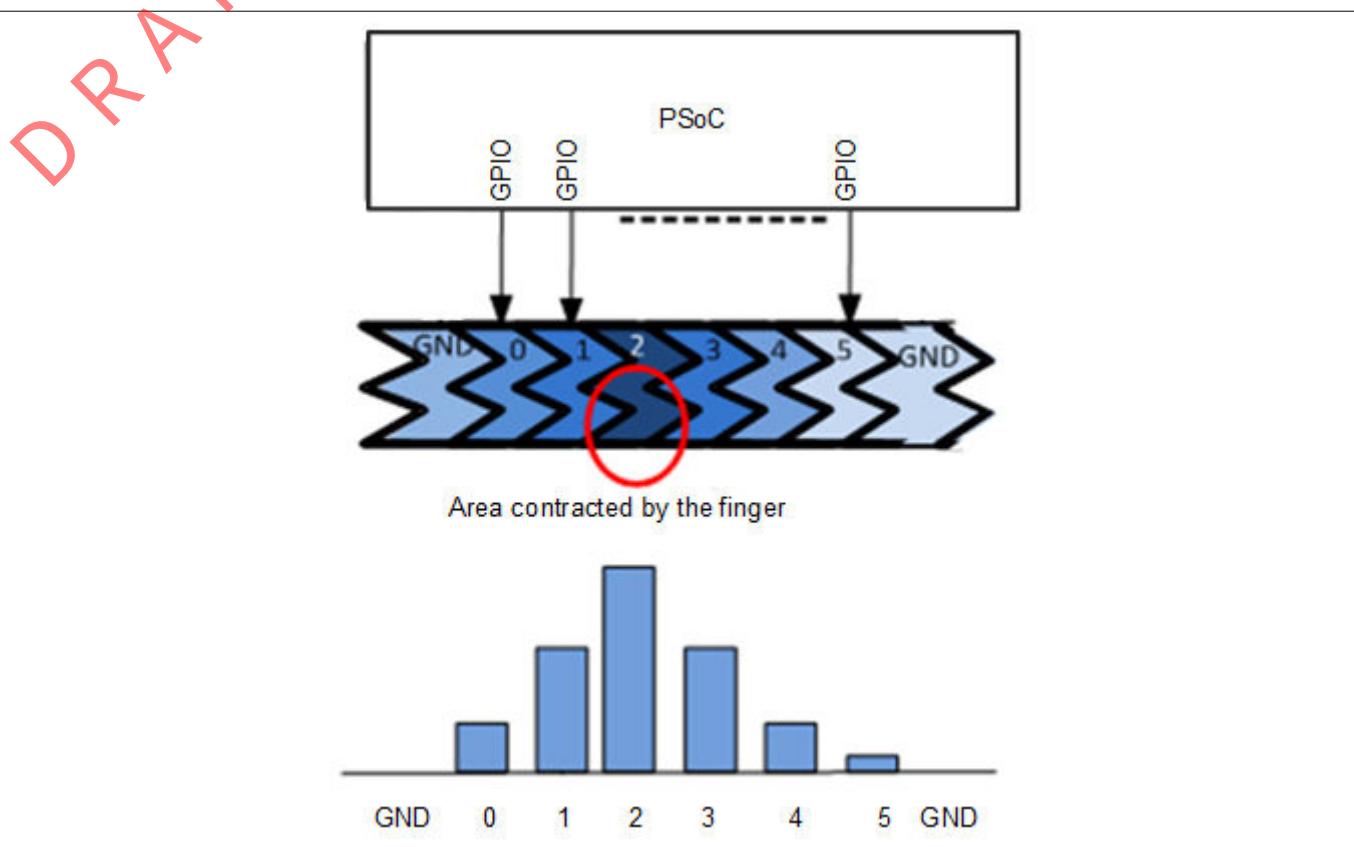


Figure 208 Linear slider

Radial sliders are similar to linear sliders except that radial sliders are continuous. [Figure 209](#) shows a typical radial slider.

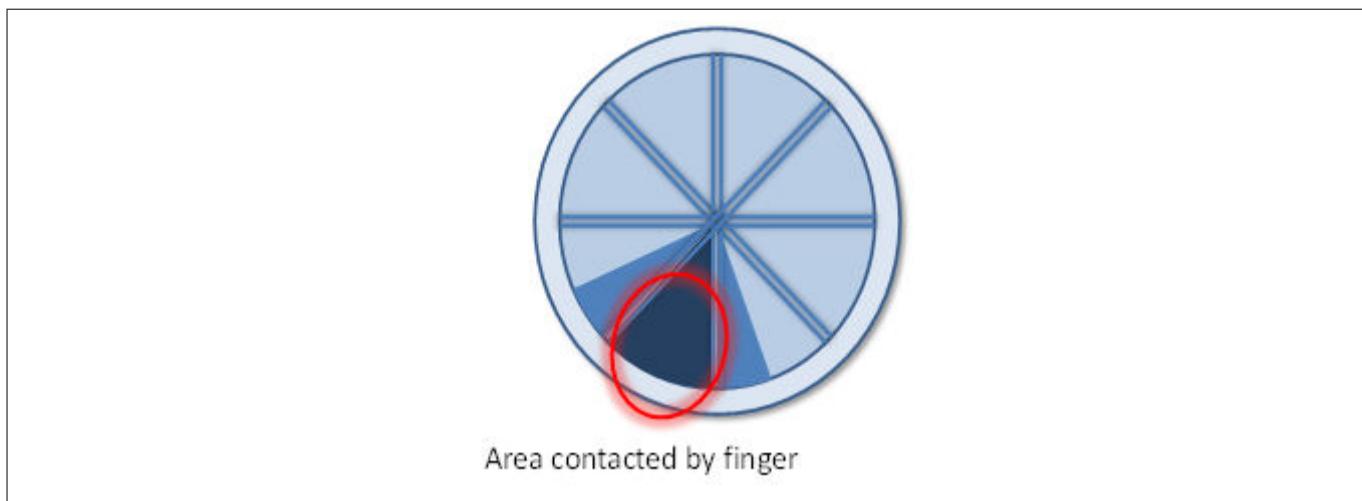


Figure 209 Radial slider

5.8.2.4.3 Touchpads/Trackpads (two-dimensional)

A touchpad (also known as trackpad) has two linear sliders arranged in an X and Y pattern, enabling it to locate a finger's position in both X and Y dimensions. [Figure 210](#) shows a typical arrangement of a touchpad sensor. Similar to the matrix buttons, touchpads can also be sensed using either CSD or CSX sensing method. CSD-based touchpads suffer from ghost touches, so it supports only single-point touch applications.

5 PSoC™ 6 application notes

CSX touchpads can support multi-point touch applications, but these may need more scanning time compared to CSD touchpad because this method scans each intersection rather than rows and columns.

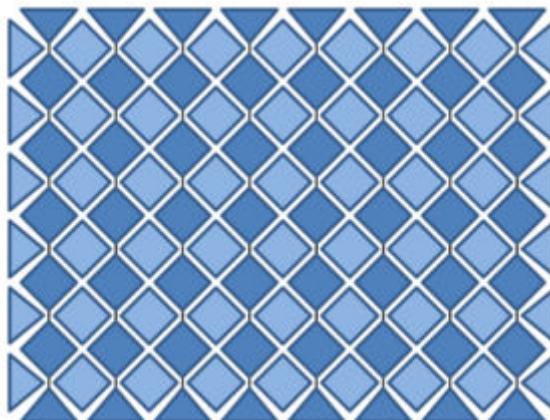


Figure 210 Touchpad sensor arrangement

5.8.2.4.4 Proximity (three-dimensional)

Proximity sensors detect the presence of a hand in the three-dimensional space around the sensor. However, the actual output of the proximity sensor is an ON/OFF state similar to a CAPSENSE™ button. Proximity sensing can detect a hand at a distance of several centimeters to tens of centimeters depending on the sensor construction. Self capacitance is the recommended method of sensing for a proximity application.

Proximity sensing requires electric fields that are projected to much larger distances than buttons and sliders. This demands a large sensor area. However, a large sensor area also results in a large parasitic capacitance C_P , and detection becomes more difficult. This requires a sensor with high electric field strength at large distances while also having a small area. [Figure 211](#) shows a proximity sensor using a trace with a thickness of 2-3 mm surrounding the other sensors.

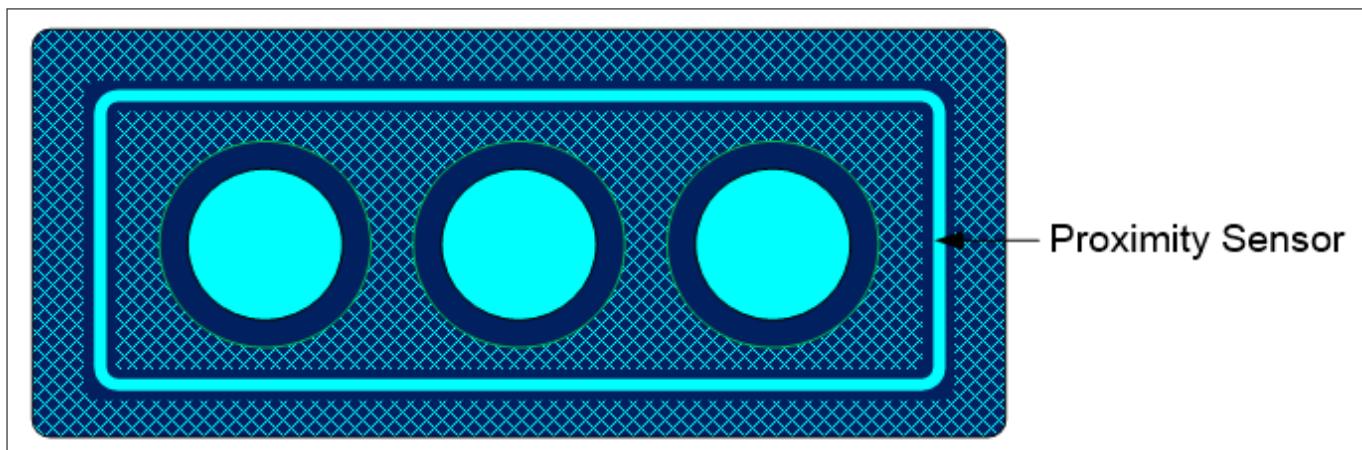


Figure 211 Proximity sensor

You can also implement a proximity sensor by ganging other sensors together. This is accomplished by combining multiple sensor pads into one large sensor using firmware. The disadvantage of this method is high parasitic capacitance. See the [Component datasheet/middleware document](#) for details on maximum parasitic capacitance supported by a given device.

See [AN92239 proximity sensing with CAPSENSE™](#) and the proximity sensing section in [Getting started with CAPSENSE™ design guide](#) to learn more about proximity sensors.

~~DRAFT~~ 5 PSoC™ 6 application notes

5.8.2.5 Liquid tolerance

Capacitive sensing is used in a variety of applications such as home appliances, automotive, and industrial applications. These applications require robust capacitive-sensing operation even in the presence of mist, moisture, water, ice, humidity, or other liquids. In a capacitive-sensing application design, false sensing of touch or proximity detection may happen due to the presence of a film of liquid or liquid droplets on the sensor surface, due to the conductive nature of some liquids. CSD sensing method can compensate for variation in raw count due to these causes and provide a robust, reliable, capacitive sensing application operation.



Figure 212 Liquid-tolerant CAPSENSE™-based touch user interface in washing machine

- To compensate for changes in raw count due to mist, moisture, and humidity changes, the CAPSENSE™ sensing method continuously adjusts the baseline of the sensor to prevent false triggers
- To prevent sensor false triggers due to a liquid flow, you should implement a [Guard sensor](#) as [Figure 213](#) shows. The [Driven shield signal and shield electrode](#) can be used to detect the presence of a streaming liquid and ignore the status or stop the sensing from rest of the sensors as long as the liquid flow is present
- Note: *the guard sensor itself is just another self-capacitance sensor; even though you could implement it around mutual-capacitance sensors also for liquid flow tolerance. PSoC™ devices allow implementation of such self-capacitance sensors and mutual-capacitance sensors together in the same design*
- To compensate for changes in raw count due to liquid droplets for self-capacitance sensing, you can implement a [Driven shield signal and shield electrode](#) as [Figure 213](#) shows. When a shield electrode is implemented, CAPSENSE™ reliably works and reports the sensor ON/OFF status correctly, even when liquid droplets are present on the sensor surface. To prevent sensor false triggers due to liquid droplets for mutual-capacitance sensing, you can use both the sensing methods that is, mutual capacitance and self-capacitance with [Driven shield signal and shield electrode](#) on the same set of sensors as [Using self-capacitance sensing for liquid tolerance of mutual-capacitance sensors](#) explains

In summary, if your application requires tolerance to liquid droplets, implement a [Driven shield signal and shield electrode](#). If your application requires tolerance to streaming liquids along with liquid droplets, implement a [Driven shield signal and shield electrode](#) and a [Guard sensor](#) as shown in [Figure 213](#). Follow the schematic and layout guidelines explained in the [Layout guidelines for liquid tolerance](#) section to construct the shield electrode and guard sensor respectively.

5 PSoC™ 6 application notes

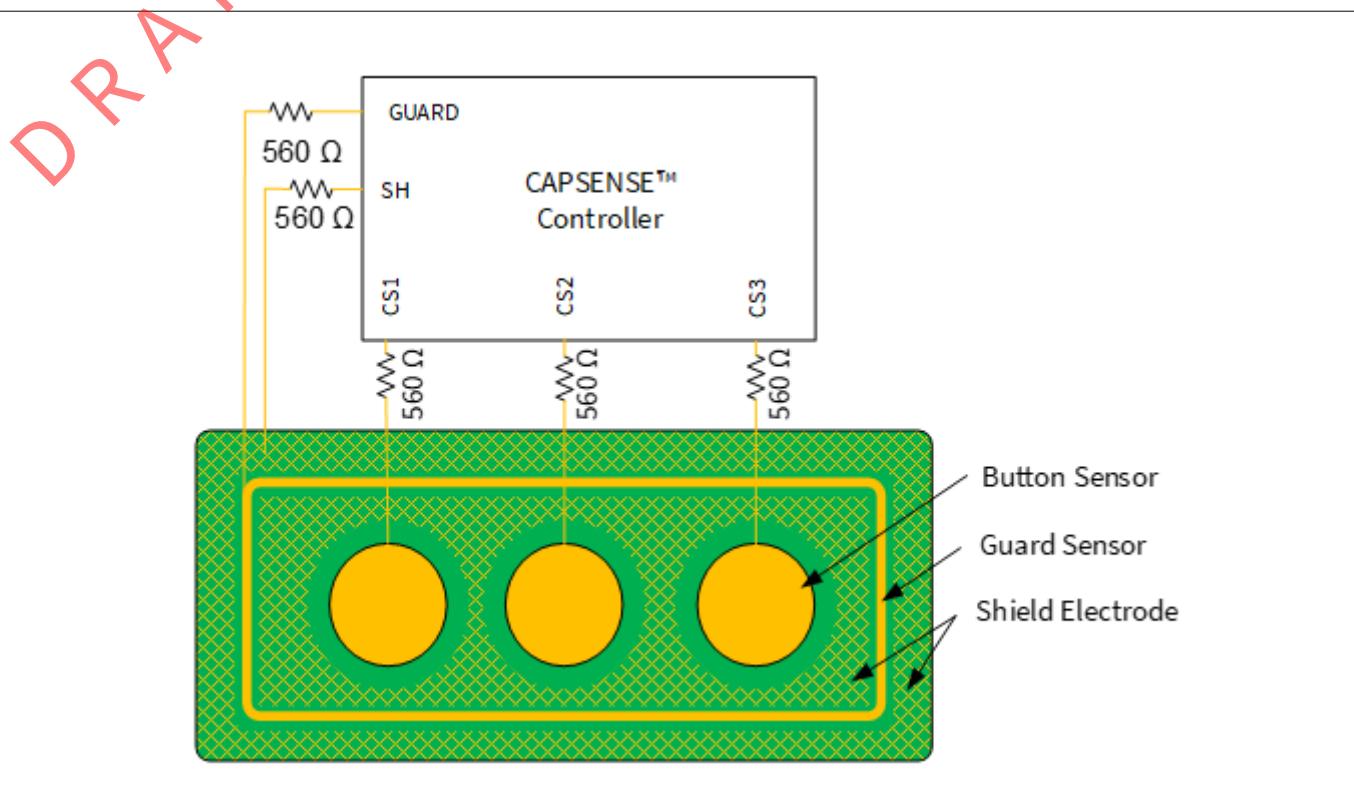


Figure 213 Shield electrode (SH) and guard sensor (GUARD) connected to CAPSENSE™ controller

5.8.2.5.1 Liquid tolerance for self capacitance sensing

Effect of liquid droplets and liquid stream on a self-capacitance sensor

To understand the effect of liquids on a CAPSENSE™ sensor, consider a CAPSENSE™ system in which the hatch fill around the sensor is connected to ground, as [Figure 214\(a\)](#) shows. The hatch fill when connected to a GND improves the noise immunity of the sensor. Parasitic capacitance of the sensor is denoted as C_p in [Figure 214\(b\)](#).

5 PSoC™ 6 application notes

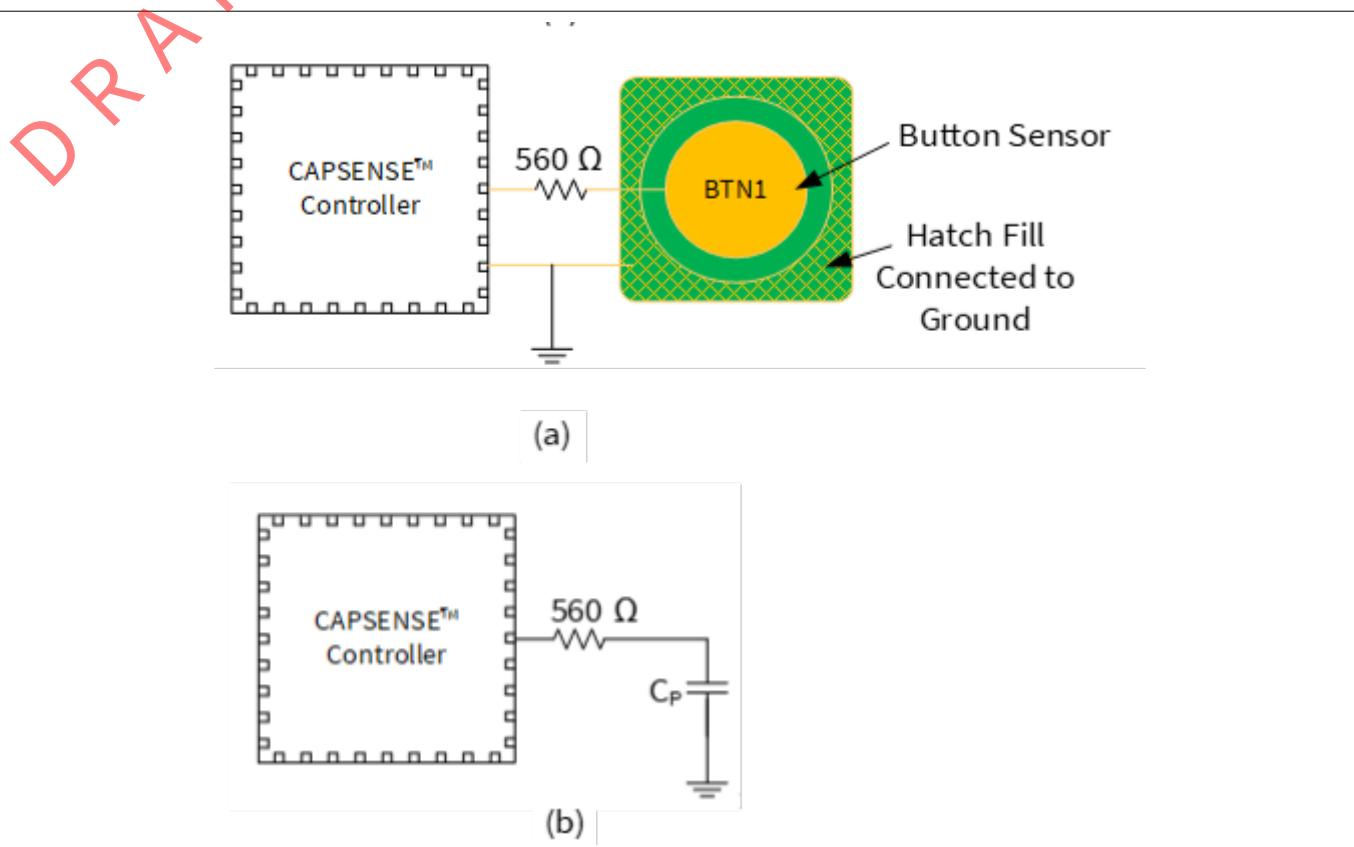


Figure 214 **Typical CAPSENSE™ system layout**

As shown in [Figure 215](#), when a liquid droplet falls on the sensor surface, due to its conductive nature it provides a strong coupling path for the electric field lines to return to ground; this adds a capacitance C_{LD} in parallel to C_P . This added capacitance draws an additional charge from the AMUX bus as explained in [GPIO cell capacitance to current converter](#) resulting in an increase in the sensor raw count. In some cases (such as salty water or water containing minerals), the increase in raw count when a liquid droplet falls on the sensor surface may be equal to the increase in raw count due to a finger touch, as [Figure 215](#) shows. In such a situation, sensor false triggers might occur.

5 PSoC™ 6 application notes

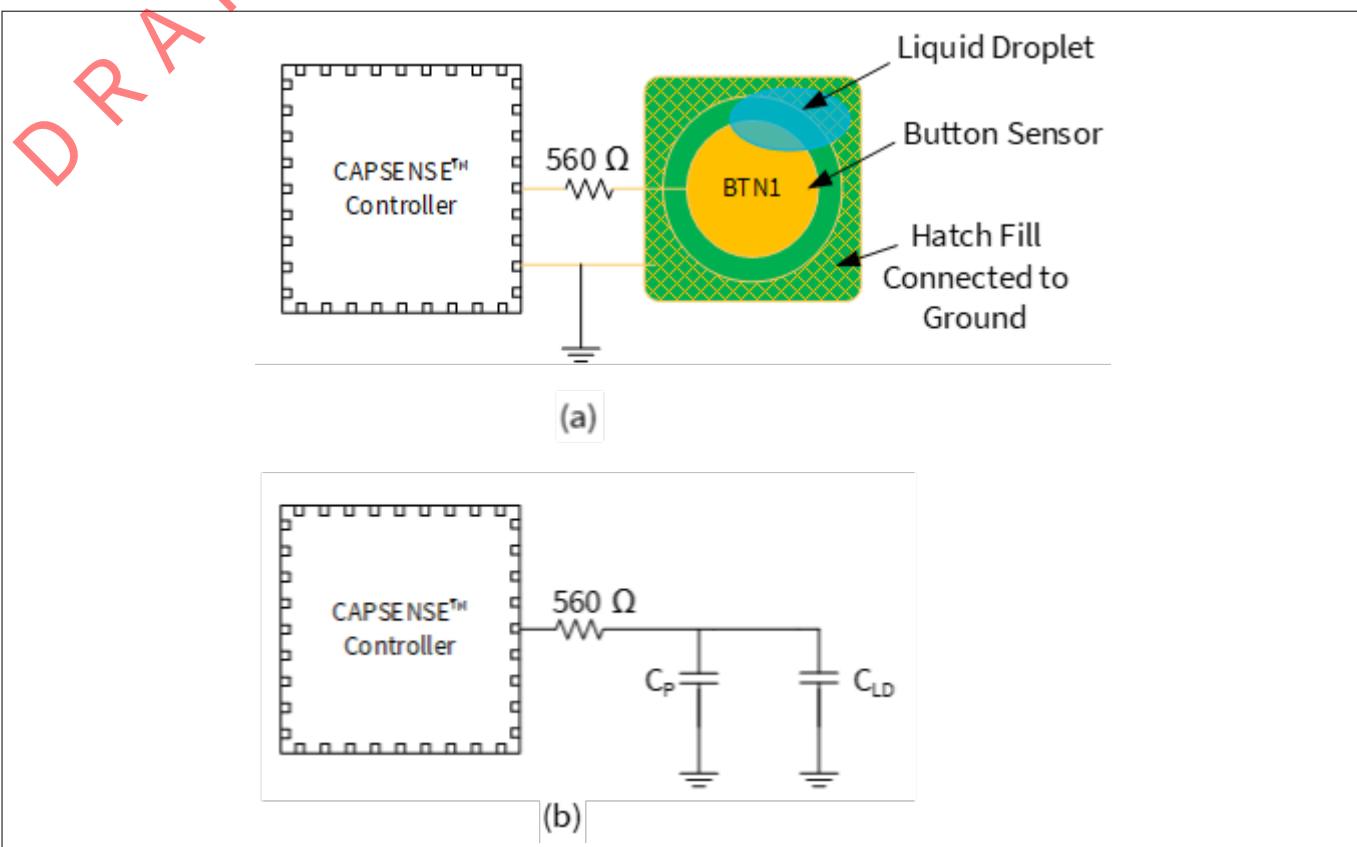


Figure 215 Capacitance added by liquid droplet when the Hatch Fill is connected to GND

C_P = Sensor parasitic capacitance

C_{LD} = Capacitance added by the liquid droplet

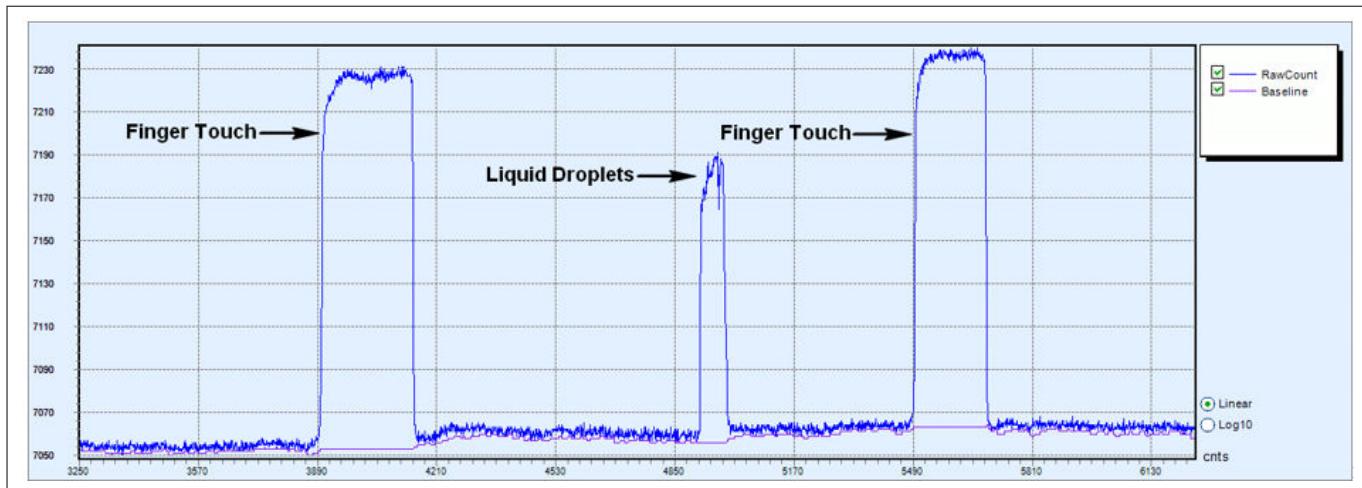


Figure 216 Effect of liquid droplet when the Hatch Fill around the sensor is connected to GND

To nullify the effect of capacitance added by the liquid droplet to the CAPSENSE™ circuitry, you should drive the hatch fill around the sensor with the driven-shield signal.

As [Figure 217](#) shows, when the hatch fill around the sensor is connected to the driven-shield signal and when a liquid droplet falls on the touch interface, the voltage on both sides of the liquid droplet remains at the same potential. Because of this, the capacitance, C_{LD} , added by the liquid droplet does not draw any additional charge from the AMUX bus and hence the effect of capacitance C_{LD} is nullified. Therefore, the increase in raw count when a water droplet falls on the sensor will be very small, as [Figure 218](#) shows.

5 PSoC™ 6 application notes

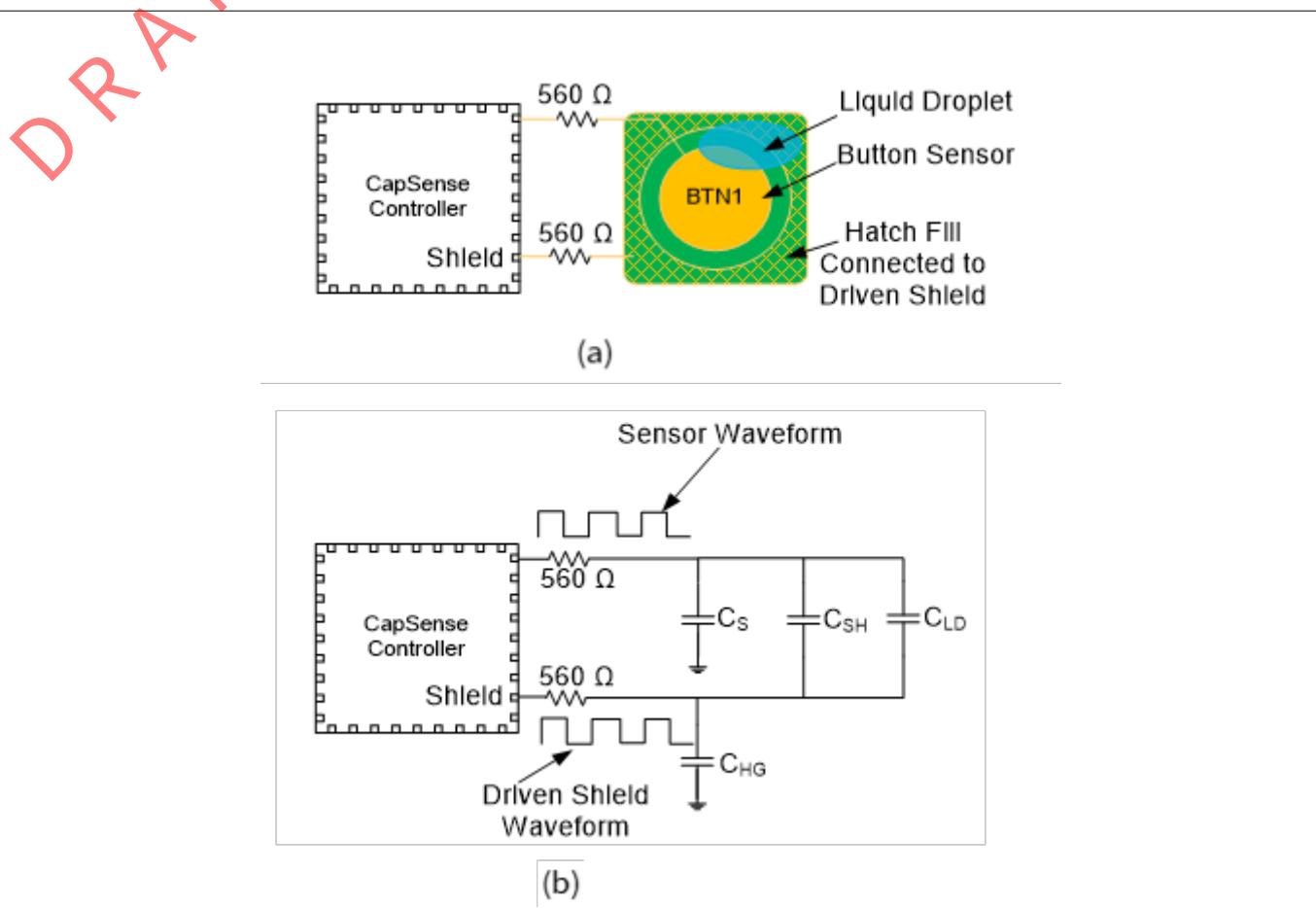


Figure 217 Capacitance added by liquid droplet when the hatch fill around the sensor is connected to shield

C_S = Sensor parasitic capacitance

C_{SH} = Capacitance between the sensor and the hatch fill

C_{HG} = Capacitance between the hatch fill and ground

C_{LD} = Capacitance added by the liquid droplet

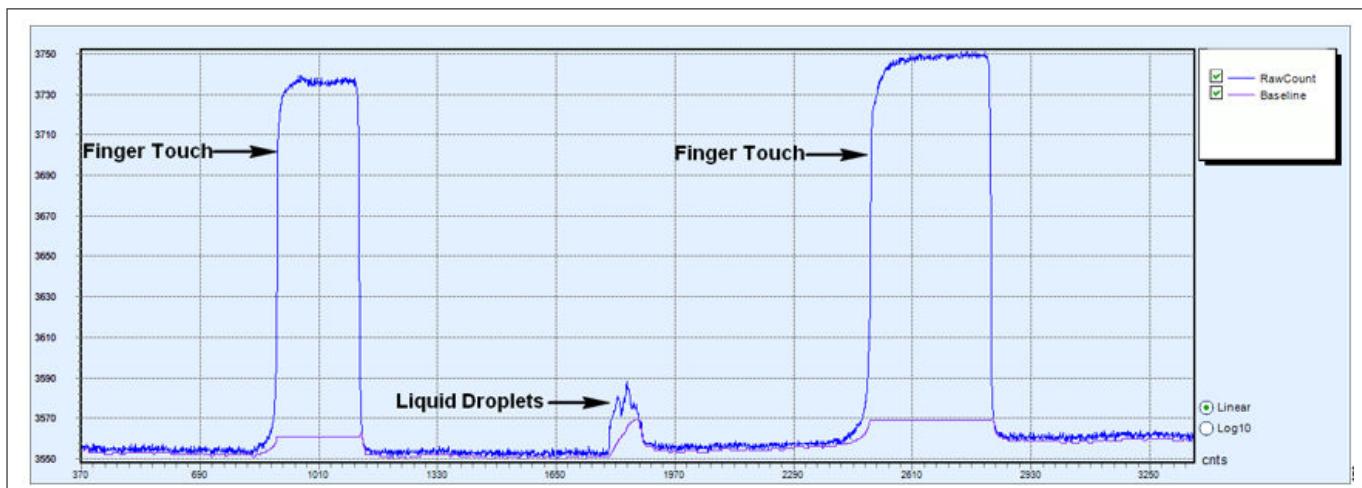


Figure 218 Effect of liquid droplet when the hatch fill around the sensor is connected to the driven-shield

~~5 PSoC™ 6 application notes~~

Figure 216 shows how a sensor may false trigger in presence of a liquid, if hatch fill is connected to ground. Note however, that the same is not true for all cases. For example, [spring sensors](#), which are inherently more liquid tolerant than sensors etched on PCB surface. As [Figure 219](#) shows, due to the large airgap between the liquid drop and the hatch fill, the capacitance C_{LD} between the liquid drop and grounded hatch pattern on the PCB would be very low so as not to cause any false triggers. If required, the hatched pattern on the PCB can still be connected to a driven shield electrode to further nullify the effect of C_{LD} and have an improved liquid tolerance.

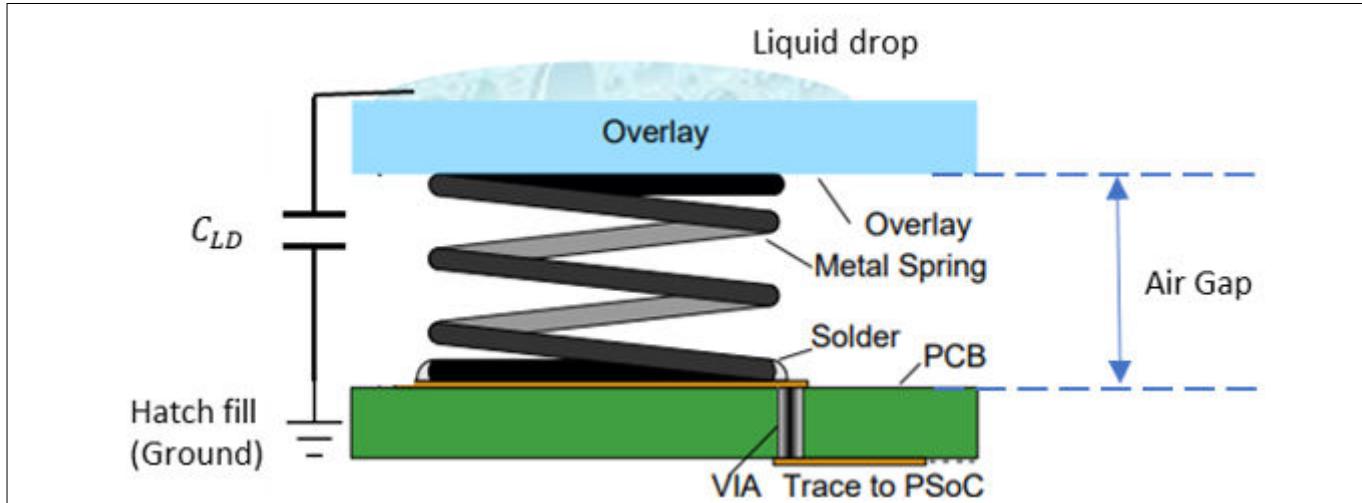


Figure 219 Capacitance added by liquid droplet in spring sensor

Driven shield signal and shield electrode

The driven-shield signal is a buffered version of the sensor-switching signal, as [Figure 220](#) shows. The driven-shield signal has the same amplitude, frequency, and phase as that of sensor switching signal. When the hatch fill around the sensor is connected to the driven shield signal, it is referred as shield electrode.

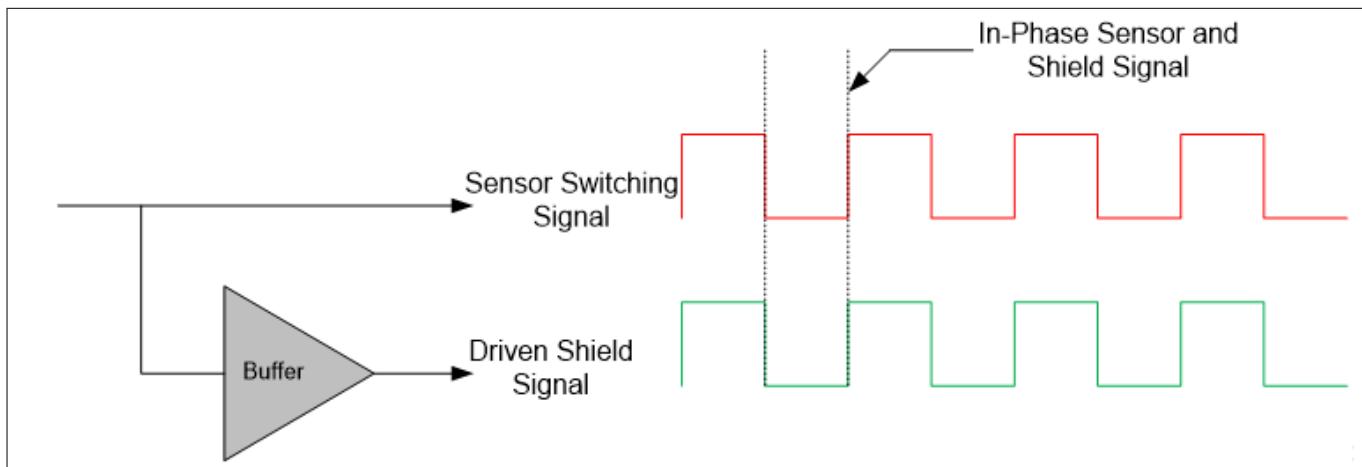


Figure 220 Driven shield signal

- To implement liquid-tolerant CAPSENSE™ designs: Shield electrode helps in making CAPSENSE™ designs liquid-tolerant as explained [above](#)
- To improve proximity sensing distance in presence of floating or grounded conductive objects: A shield electrode, when placed between the proximity sensor and a floating or a grounded conductive object, reduces the effect of these objects on the proximity-sensing distance and helps in achieving large

5 PSoC™ 6 application notes

~~DO NOT USE~~ proximity-sensing distance. See the “Proximity Sensing” section in the [Getting started with CAPSENSE™ design guide](#) for more details

- To reduce the parasitic capacitance of the sensor: When a CAPSENSE™ sensor has a long trace, the CP of the sensor will be very high because of the increased coupling of sensor electric field lines from the sensor trace to the surrounding ground. By implementing a shield electrode, the coupling of electric field lines to ground is reduced, which results in reducing the CP of the sensor

See [Layout guidelines for shield electrode](#) for layout guidelines of shield electrode.

Guard sensor

When a continuous liquid stream is present on the sensor surface, the liquid stream adds a large capacitance (C_{ST}) to the CAPSENSE™ sensor. This capacitance may be several times larger than C_{LD} . Because of this, the effect of the shield electrode is completely masked, and the sensor raw counts will be same as or even higher than a finger touch. In such situations, a guard sensor is useful to prevent sensor false triggers.

A guard sensor is a copper trace that surrounds all the sensors on the PCB, as [Figure 221](#) shows. A guard sensor is similar to a button sensor and is used to detect the presence of streaming liquids. When a guard sensor is triggered, the firmware should disable the scanning of all other sensors except the guard sensor to prevent sensor false triggers.

Note: *The sensors are not scanned, or the sensor status is ignored when the guard sensor is triggered; therefore, touch cannot be detected when there is a liquid stream on the touch surface.*

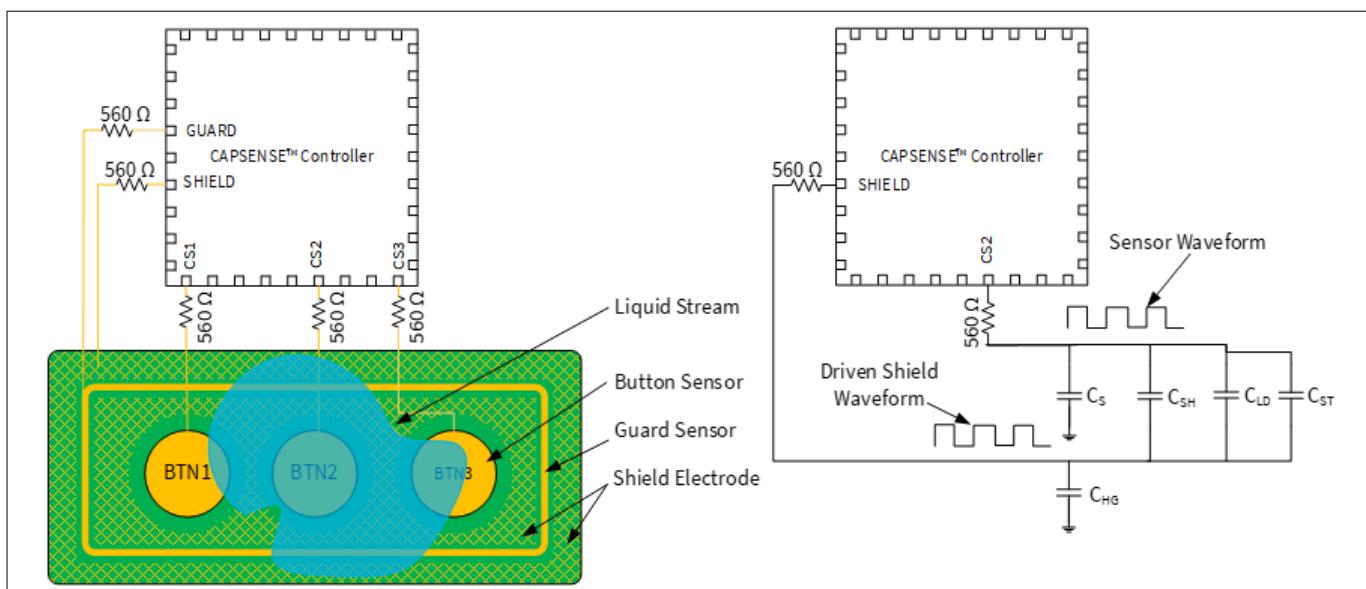


Figure 221 **Measurement with a liquid stream**

See [Layout guidelines for guard sensor](#) for PCB layout guidelines for implementing a guard sensor.

If there is no space on the PCB for implementing a guard sensor, the guard sensor functionality can be implemented in the firmware. For example, you can use the ON/OFF status of different sensors to detect a liquid stream depending on the use case, such as follows:

- When there is a liquid stream, more than one button sensor will be active at a time. If your design does not require multi-touch sensing, you can detect this and ignore the sensor status of all the button sensors to prevent false triggering

5 PSoC™ 6 application notes

- In a slider, if the slider segments which are turned ON are not adjacent segments, you can reset the slider segments status or ignore the slider centroid value that is calculated
- Likewise, you could create your own custom algorithm to detect the presence of streaming liquids and ignore the sensor status during the time a liquid is present on the touch surface

Note: *The sensors are not scanned, or the sensor status is ignored when the guard sensor is triggered; therefore, touch cannot be detected when there is a liquid stream on the touch surface*

5.8.2.5.2 Liquid tolerance for mutual-capacitance sensing

Effect of liquid droplets and liquid stream on a mutual-capacitance sensor

Mutual-capacitance buttons often have a grounded hatch fill around the sensors for improved noise immunity. If a liquid droplet falls over the sensor while covering some part of the grounded hatch, the mutual-capacitance decreases similar to the effect of placing a finger on the sensor. This decrease in mutual-capacitance causes an increase in raw count as explained in [CAPSENSE™ CSX sensing method \(third- and fourth- generation\)](#) and as shown in the [Figure 222](#). The amount of increase in the raw count depends on the size and characteristics of the liquid drop.

However, mutual-capacitance increases if the liquid droplet covers just the Tx and Rx electrode and does not spread over the grounded hatch. This causes a decrease in raw count as shown in [Figure 222](#). This decrease in raw count may cause the baseline reset due to [Low baseline reset](#). Once the liquid drop is removed, the raw count would rise while the baseline may remain at the lower value, resulting in a difference signal which may cause the sensor to false trigger.

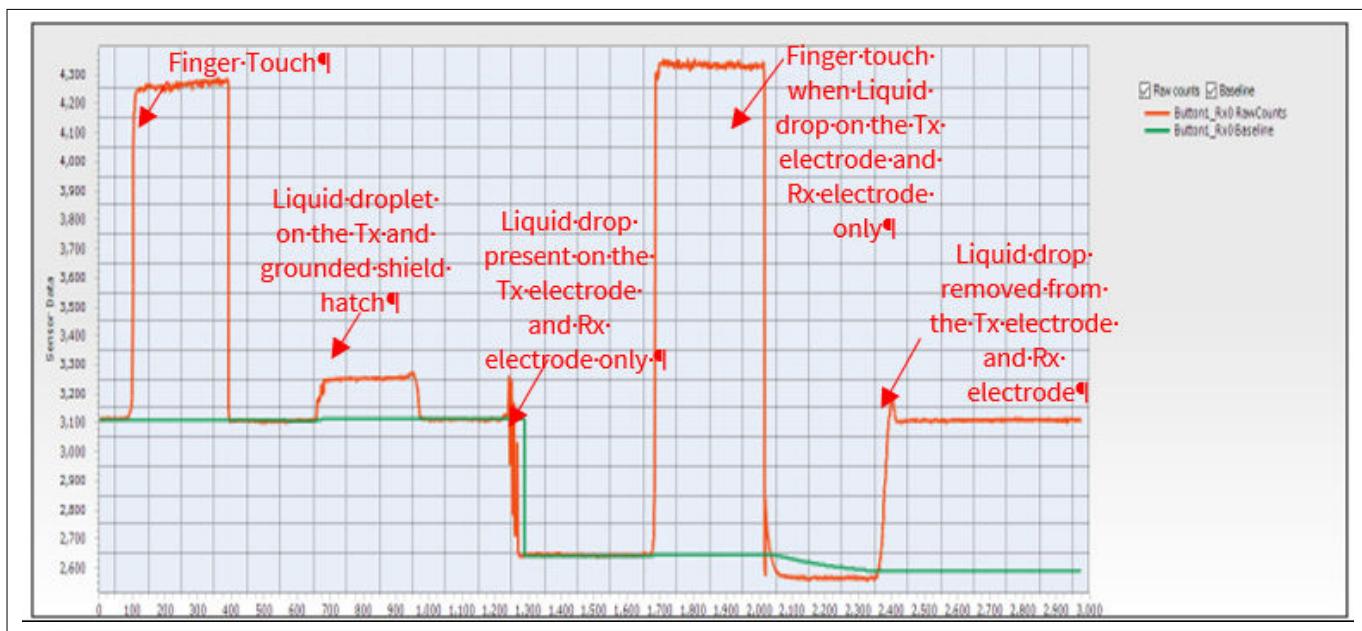


Figure 222 Effect of liquid droplet on CSX sensor when the Hatch Fill around the sensor is connected to ground

Using self-capacitance sensing for liquid tolerance of mutual-capacitance sensors

CAPSENSE™ senses the self-capacitance of Tx and Rx nodes of a mutual-capacitance sensor. This ability of scanning the sensor using both CSD and CSX modes could be used to avoid false triggers due to the presence of

5 PSoC™ 6 application notes

~~DRAFT~~
liquid drops on a mutual capacitance sensor. See the code example [PSoC™ 4 hybrid sensing using CAPSENSE™](#) to understand how to sense a mutual-capacitance button with both CSD as well as CSX sensing method.

To achieve liquid tolerance, you need to scan the Rx electrode of the sensor with the CSD sense method. While scanning the Rx electrode as a CSD sensor, ensure that you enable the shield electrode, and connect the Tx pin of the mutual-capacitance sensor to the driven shield signal. You can use the low-level API function CapSense_SetPinState() to connect the Tx pin of the mutual-capacitance sensor to the shield electrode before calling the CapSense_ScanAllWidgets() API function that scans the Rx electrode as a CSD sensor as shown below:

```
CapSense_SetPinState(CapSense_BUTTON1_WDGT_ID,CapSense_BUTTON1_TX0_ID,CapSense_SHIELD);
CapSense_ScanAllWidgets();
```

From sections [2.5.1](#) and [2.5.2](#) you understood the effect of liquid drop on the CSD and CSX button respectively. By utilizing the difference in their response to the liquid drop, you can create a firmware logic to achieve a liquid-tolerant mutual-capacitance sensor. The effect of presence of the liquid drop on the CSD and CSX scan results is summarized in [Figure 223](#).

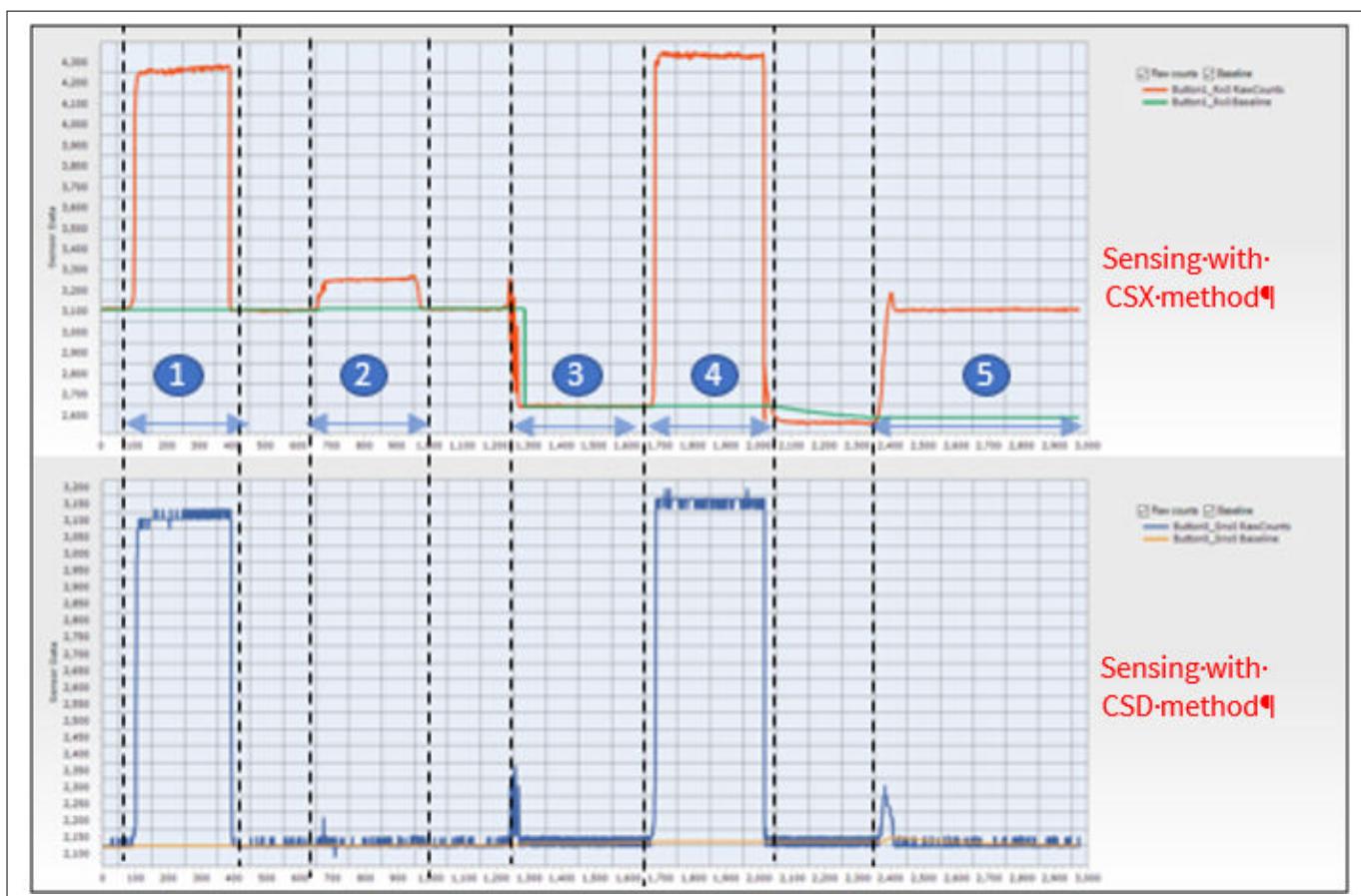


Figure 223 Effect of water drop on the CSX sensor pattern scanned with CSD and CSX methods

Where [Figure 223](#) shows the effect of the water drop on the CSX sensor pattern surrounded by hatch fill when scanned using this method. The regions in [Figure 223](#) represent the following:

1. Finger touch
2. Liquid droplet on the Tx line and grounded shield hatch
3. Liquid drop present on the Tx and Rx electrodes only
4. Finger touch when a liquid drop is on the Tx and Rx electrodes only
5. Liquid drop removed from the Tx and Rx electrodes

5 PSoC™ 6 application notes

The changes in raw count as shown in [Figure 223](#) can be used in the firmware to reset the baseline of the CSX sensor to nullify the effect of liquid drops. The button status should be ON state for Region 1, 4, and OFF state in other regions; additionally, the baseline of the CSX button must be re-initialized in Region 3 and Region 5. The baseline of the sensor could be reset by using the `CapSense_InitializeWidgetBaseline()` API function as shown below:

```
CapSense_InitializeWidgetBaseline(CapSense_CSX_BUTTON_WDGT_ID);
```

See the [Component datasheet/middleware document](#) or more details on using this API; see [Selecting CAPSENSE™ software parameters](#) to learn about the baseline of the sensor.

5.8.2.5.3 Effect of liquid properties on liquid-tolerance performance

In certain applications, the CAPSENSE™ system has to work in the presence of a variety of liquids such as soap water, sea water, and mineral water. In such applications, it is always recommended to tune the CAPSENSE™ parameters for sensors by considering the worst-case signal due to liquid droplets. To simulate the worst-case conditions, it is recommended that you test the liquid-tolerance performance of the sensors with salty water by dissolving 40 grams of cooking salt (NaCl) in one liter of water. Tests were done using soapy water; the results show that the effect of soapy water is similar to the effect of salty water. Therefore, if the tuning is done to reject salty water, the CAPSENSE™ system will work even in the presence of soapy water.

In applications such as induction cooktops, there are chances of hot water spilling on to the CAPSENSE™ touch surface. To determine the impact of the temperature of a liquid droplet on CAPSENSE™ performance, droplets of water at different temperatures were poured on a sensor and the corresponding change in raw counts was monitored. Experiment shows that the effect of hot liquid droplets is same as that of the liquid at room temperature as [Figure 224](#) shows. This is because the hot liquid droplet cools down immediately to room temperature when it falls on the touch surface. If hot water continuously falls on the sensor and the temperature of the overlay rises because of the hot water, the increase in raw count due to the increase in temperature is compensated by the [Baseline update algorithm](#), thereby preventing any false triggering of the sensors.

5 PSoC™ 6 application notes

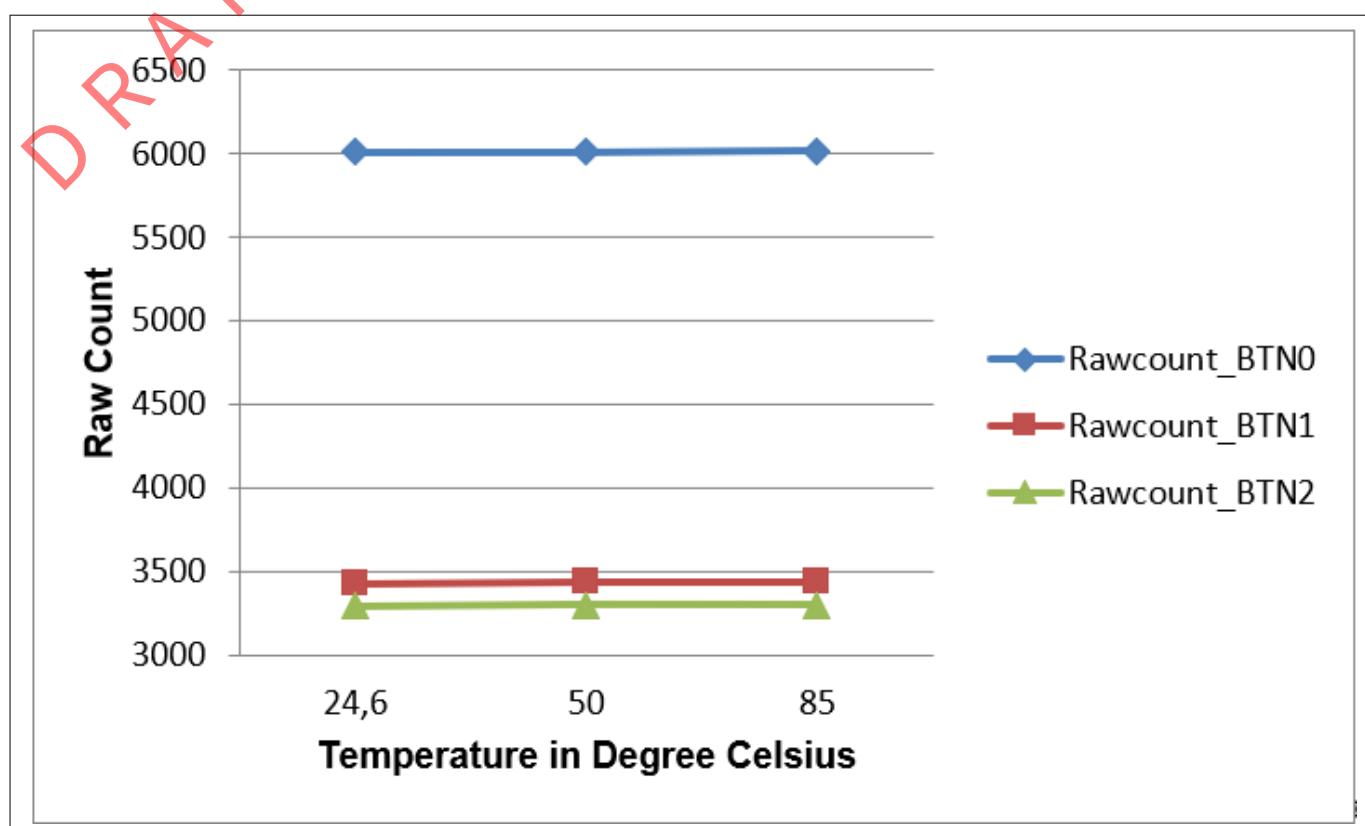


Figure 224 Raw count variation versus water temperature

~~DO NOT USE~~ 5 PSoC™ 6 application notes

5.8.3 PSoC™ 4 and PSoC™ 6 MCU CAPSENSE™

This chapter explains how CAPSENSE™ CSD and CSX (Third, Fourth, and Fifth generations) is implemented in the PSoC™ 4 and PSoC™ 6 MCU. See [Capacitive touch sensing method](#) to understand the basic principles of CAPSENSE™. A basic knowledge of the PSoC™ device architecture is a prerequisite for this chapter. If you are new to PSoC™ 4, see [AN79953 - Getting started with PSoC™ 4](#) or [AN91267 - Getting started with PSoC™ 4 Bluetooth® LE](#); for PSoC™ 6 MCU, see [AN221774 - Getting started with PSoC™ 6 MCU](#).

You can skip this chapter if you are using the automatic tuning feature (SmartSense) of the Component. See the [CAPSENSE™ performance tuning](#) chapter for details.

The PSoC™ 4 family of devices has three different CAPSENSE™ architectures. [Table 35](#) explains the differences between the Third, Fifth-Generation CAPSENSE™ architecture.

5.8.3.1 CAPSENSE™ generations in PSoC™ 4 and PSoC™ 6

[Table 35](#) lists the main differences in the CAPSENSE™ architecture.

Table 35 Comparison of CAPSENSE™ architecture for CSD and CSX

Feature	Third-generation CAPSENSE™	Fourth-generation CAPSENSE™	Fifth-generation CAPSENSE™	Improvement impact	Conditions
SNR	5:1	6.5:1	48:1	Higher SNR implies better sensitivity, that is ability to sense smaller signal.	$V_{DD} = 5\text{ V}$; No firmware filter; $C_p \approx 33\text{ pF}$; $C_f = 0.1\text{ pF}$
Sensing mode	Self-cap and Mutual-cap modes	Self-cap, Mutual-cap modes and ADC modes	Self-cap and Mutual-cap modes	–	–
Sensor capacitance parasitic range	5 pF – 45 pF	5 pF – 200 pF	2 pF – 200 pF	Greater C_p range implies higher flexibility in PCB layout routing and ability to sense with very short/long sensor traces, and for different PCB materials (for example: FFC and so on).	–
Typical sense signal needed	100 fF	100 fF	15 fF for CSD-RM 10 fF for CSX-RM	Smaller sense signal required, implying support for thicker overlays, higher proximity range, smaller sensor size and so on.	$V_{DD} = 5\text{ V}$; No firmware filter; $C_p \approx 33\text{ pF}$; SNR = 5:1;

(table continues...)

DRAFT

5 PSoC™ 6 application notes

Table 35 (continued) Comparison of CAPSENSE™ architecture for CSD and CSX

Feature	Third-generation CAPSENSE™	Fourth-generation CAPSENSE™	Fifth-generation CAPSENSE™	Improvement impact	Conditions
Noise floor (rms)	–	–	500 aF for CSD-RM 100 aF for CSX-RM	Higher SNR or lower noise floor implies ability to sense smaller signal.	$V_{DD} = 5\text{ V}$; $C_p \approx 33\text{ pF}$; $C_M = 5\text{ pF}$
Overlay thickness supported	Up to 5 mm	Up to 5 mm	Up to 18 mm	Supports designs with thicker overlay.	10 mm CSD button; Acrylic overlay; SNR = 5:1; $C_p \approx 22\text{ pF}$;
Refresh rate	–	22 Hz	242 Hz	Faster refresh rate enables fast gestures and taps detections on applications such as large trackpad and long sliders or large number of button sensors with single device, and so on.	7x5 CSX touchpad; Acrylic overlay 3 mm thickness; SNR = 10:1; Finger Size = 8 mm;
CPU bandwidth requirement	Completely CPU driven. CPU is required for initialization and sequencing the sensors.	40% Sequencer ¹⁵ takes care of initialization, configuration and scanning of sensors. CPU needed for sequencing through each sensor.	7% Completely autonomous.	Reduced CPU usage for sensing, frees CPU to perform other peripheral operations and act as a central controller in an application.	10x8 CSX touchpad; Scan clock = 1 MHz; No of sub-conversions = 70; Refresh rate = 100 Hz;
Emission control options.	PRS	PRS, SSC	PRS, SSC	–	–
Noise immunity	Sense Voltage (V_{ref})	1.2 V	1.2 V-2.8 V.	Rail to Rail	Higher the sense voltage, higher the noise immunity.

(table continues...)

¹⁵ The hardware state machine is a logic which controls the CAPSENSE™ block and sensor scanning.

5 PSoC™ 6 application notes

Table 35 (continued) Comparison of CAPSENSE™ architecture for CSD and CSX

Feature	Third-generation CAPSENSE™	Fourth-generation CAPSENSE™	Fifth-generation CAPSENSE™	Improvement impact	Conditions
Differenti al Sensing	Mutual-Cap sensing	Mutual-Cap sensing	Mutual-Cap and Self-Cap sensing	Differential sensing cancels out noise induced from external environment through C_{MOD} .	
	V_{DD} noise impact	Yes	Yes	V_{DD} noise have minimal affect on fifth generation CAPSENSE™ operation.	
Sense clock frequency	Self-Cap	45 kHz – 6 MHz	45 kHz – 6 MHz	45 kHz – 6 MHz	Higher sense clock frequency means faster scan for low C_p sensors. This provides ability to support faster taps or gestures, or for a given refresh rate, ability to implement multiple firmware filters for better immunity.
	Mutual-Cap	45 kHz - 300 kHz	45 kHz - 3 MHz	45 kHz – 6 MHz	
Multi-channel support	No	No	Yes	Provides 'n' times increased speed of scanning for the same number of sensors, if 'n' channels are used.	-
Shield Cp	--	--	1.2 nF	-	-
Device family	PSoC™ 4100/4200 PSoC™ 4100 M/4200 M PSoC™ 4100 L/4200 L PSoC™ 4100 BL/4200 BL	PSoC™ 4000 PSoC™ 4000S PSoC™ 4100S PSoC™ 4100S Plus PSoC™ 6	PSoC™ 4100 S Max	-	-

5.8.3.2 CAPSENSE™ CSD sensing method (third and fourth generation)

Figure 225 illustrates the CAPSENSE™ block that scans CAPSENSE™ sensors in CSD sensing mode.

5 PSoC™ 6 application notes

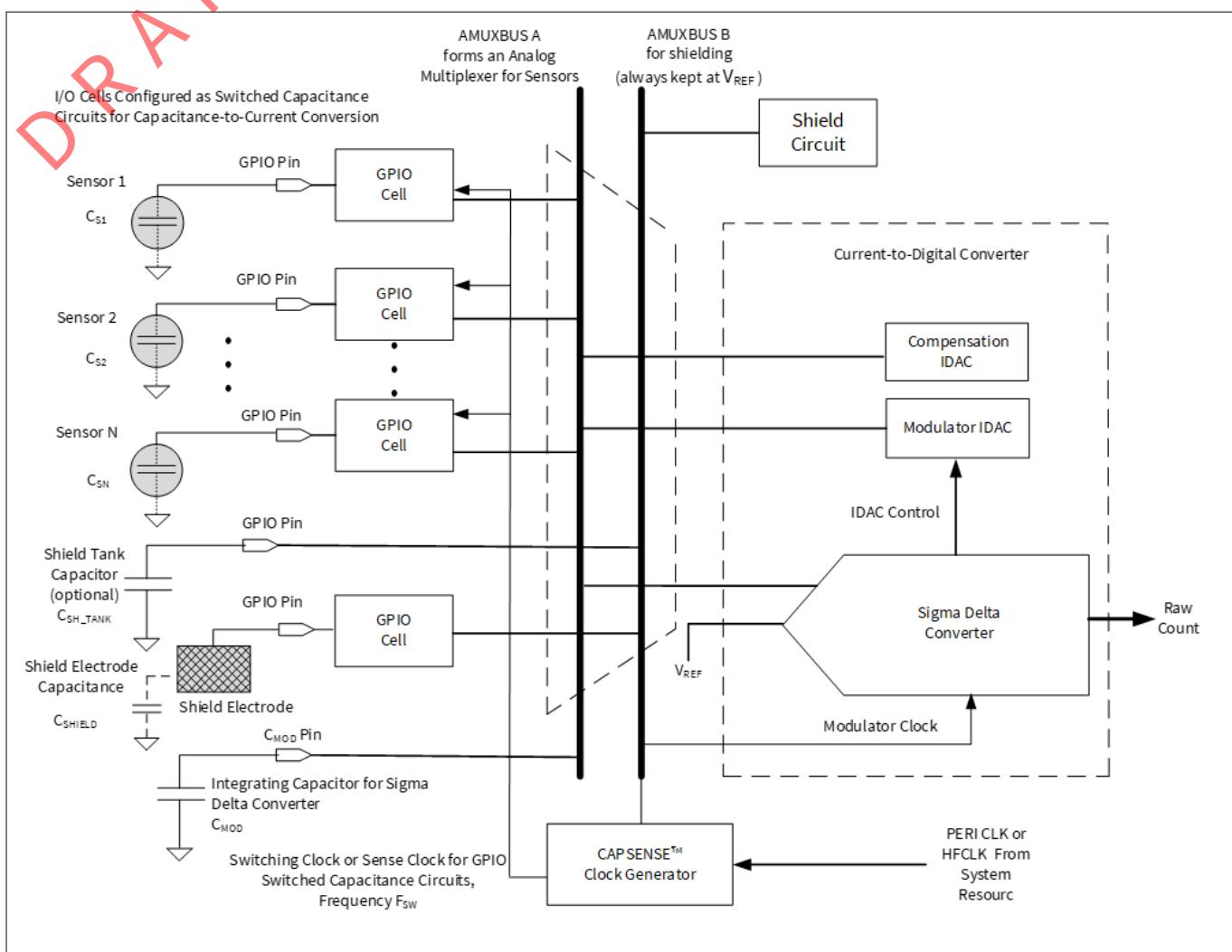


Figure 225 CAPSENSE™ CSD sensing

As explained in [Capacitive touch sensing method](#), this block works by first converting the sensor capacitance into an equivalent current. An analog multiplexer then selects one of the currents and feeds it into the current-to-digital converter. This current-to-digital converter consists of a sigma-delta converter, which controls the modulation IDAC for a specific period, the total current sourced or sunk by the IDACs is the same as the total current sunk or sourced by the sensor capacitance. The digital count output of the sigma-delta converter is an indicator of the sensor capacitance and is called a raw count. This block can be configured in either IDAC Sourcing mode or IDAC Sinking mode. In the IDAC Sourcing mode, the IDACs source current to AMUXBUS while the GPIO cells sink current from AMUXBUS. In the IDAC Sinking mode, the IDACs sink current from AMUXBUS while the GPIO cells source current to AMUXBUS.

5.8.3.2.1 GPIO cell capacitance to current converter

In the CAPSENSE™ CSD system, the GPIO cells are configured as switched-capacitance circuits that convert sensor capacitances into equivalent currents. [Figure 226](#) shows a simplified diagram of the GPIO cell structure.

5 PSoC™ 6 application notes

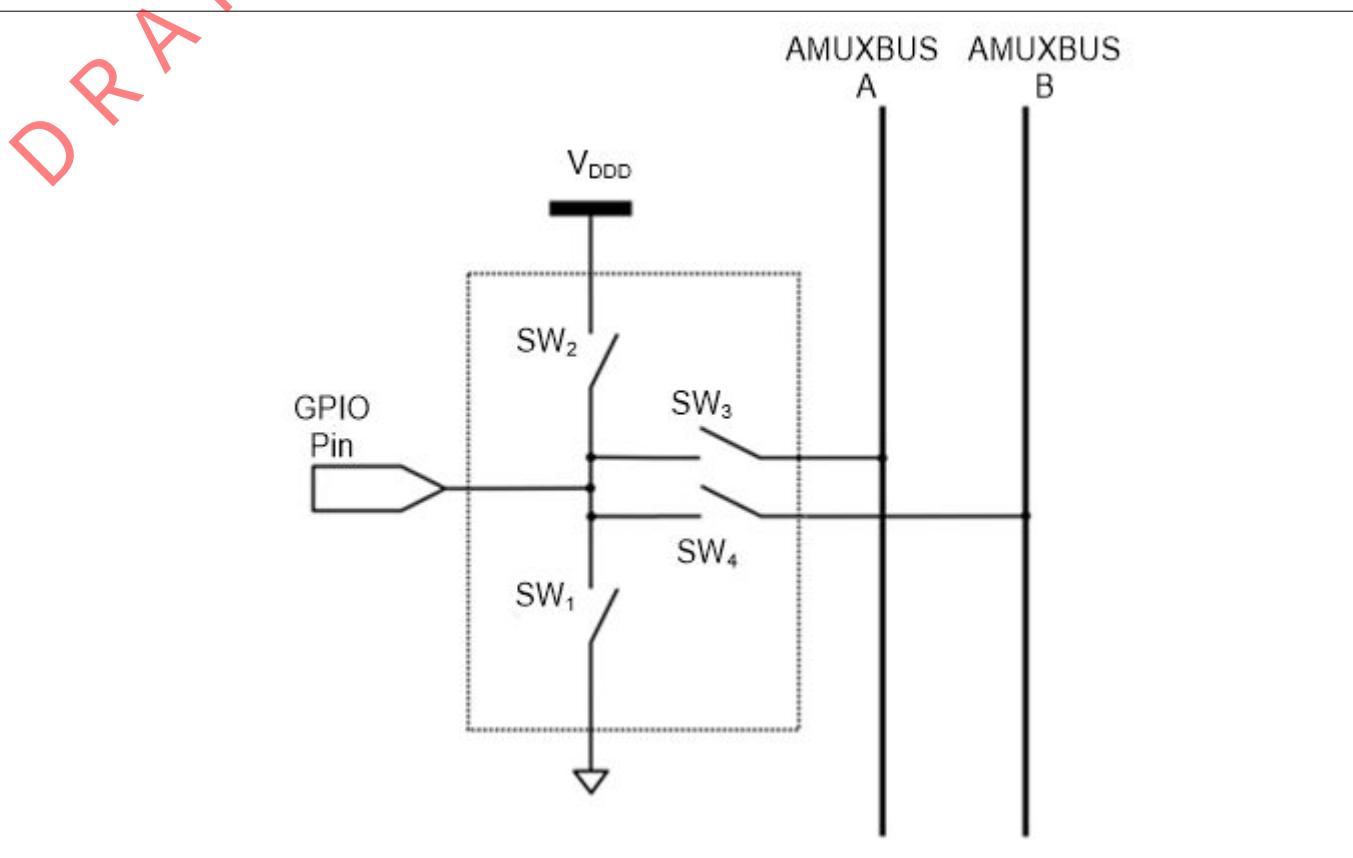


Figure 226 **GPIO cell structure**

PSoC™ 4 and PSoC™ 6 devices consist of two AMUX buses: AMUXBUS A is used for CSD sensing and AMUXBUS B is used for CAPSENSE™ CSD shielding. The GPIO switched-capacitance circuit has two possible configurations: source current to AMUXBUS A or sink current from AMUXBUS A

5.8.3.2.2 IDAC sourcing mode

In the IDAC Sourcing mode, the GPIO cell sinks current from the AMUXBUS A through a switched capacitor circuit as Figure 227 shows.

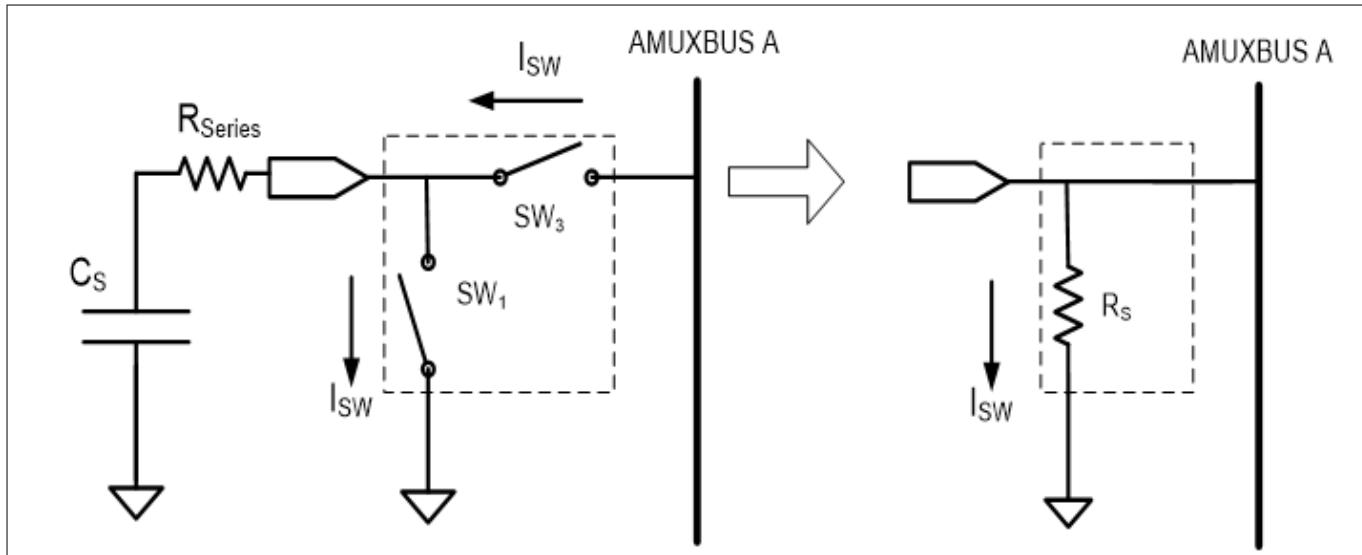


Figure 227 **GPIO cell sinking current from AMUXBUS A**

5 PSoC™ 6 application notes

Two non-overlapping, out-of-phase clocks of frequency f_{SW} control the switches SW_1 and SW_3 as Figure 228 shows. The continuous switching of SW_1 and SW_3 forms an equivalent resistance R_S , as Figure 227 shows.

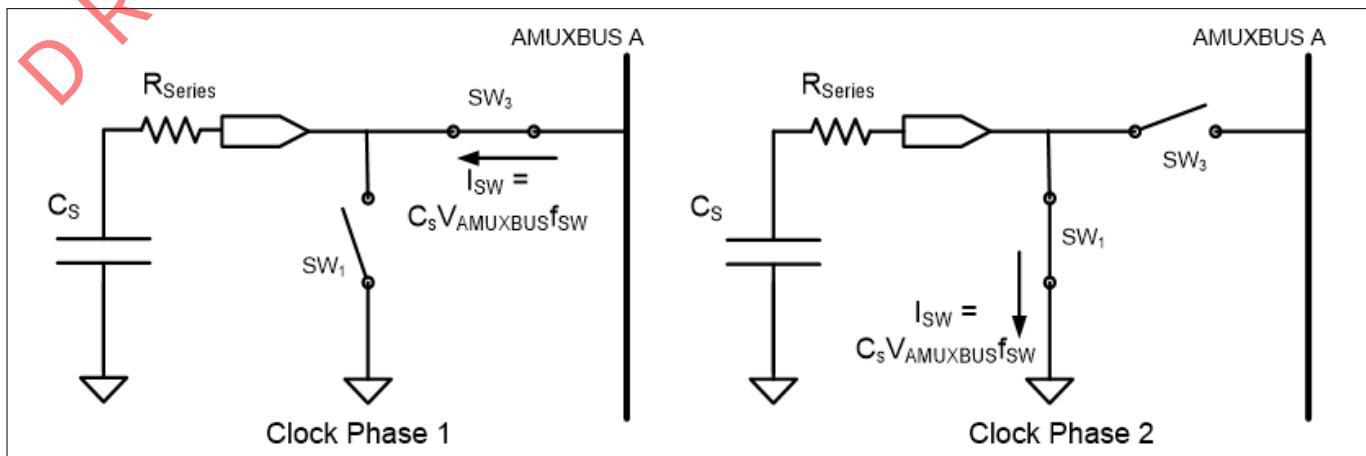


Figure 228 **SW_1 and SW_3 switch in non-overlapping manner**

If the switches operate at a sufficiently low frequency f_{SW} , such that time $T_{SW}/2$ is sufficient to fully charge the sensor to V_{REF} and fully discharge it to ground, as Figure 228 shows, the value of the equivalent resistance R_S is given by Equation 6.

$$R_S = \frac{1}{C_S f_{SW}}$$

Equation 6 **Sensor equivalent resistance**

Where,

C_S = Sensor capacitance

f_{SW} = Frequency of the sense clock

The sigma-delta converter maintains the voltage of AMUXBUS A at a constant V_{REF} (this process is explained in Sigma-delta converter. Figure 229 shows the resulting voltage waveform across C_S .

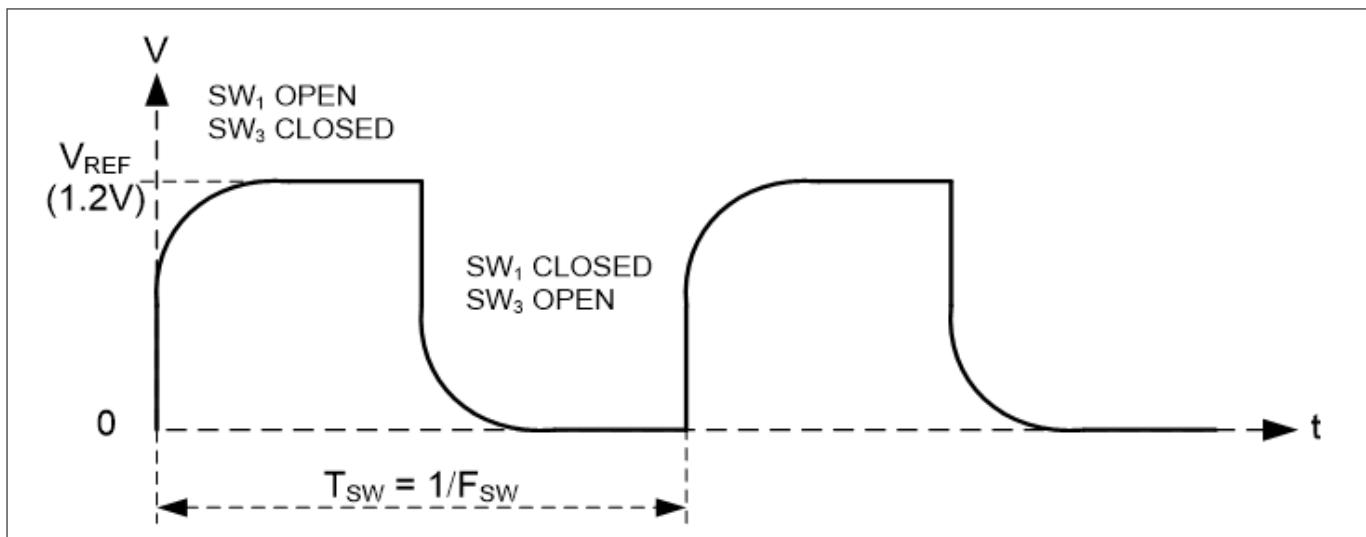


Figure 229 **Voltage across sensor capacitance**

Equation 7 gives the value of average current taken from AMUXBUS A.

5 PSoC™ 6 application notes

$$I_{CS} = C_S F_{SW} V_{REF}$$

~~DRAFT~~

Equation 7 Average current sunk from AMUXBUS A to GPIO through CAPSENSE™ sensor (ICS)

5.8.3.2.3 IDAC sinking mode

In the IDAC sinking mode, the GPIO cell sources current to the AMUXBUS A through a switched capacitor circuit as [Figure 230](#) shows. [Figure 231](#) shows the voltage waveform across the sensor capacitance.

Because this mode charges the AMUXBUS A directly through V_{DDD} , it is more susceptible to power supply noise compared to the IDAC sourcing mode. Hence, it is recommended to use this mode with an LDO or a very stable and quiet V_{DDD} .

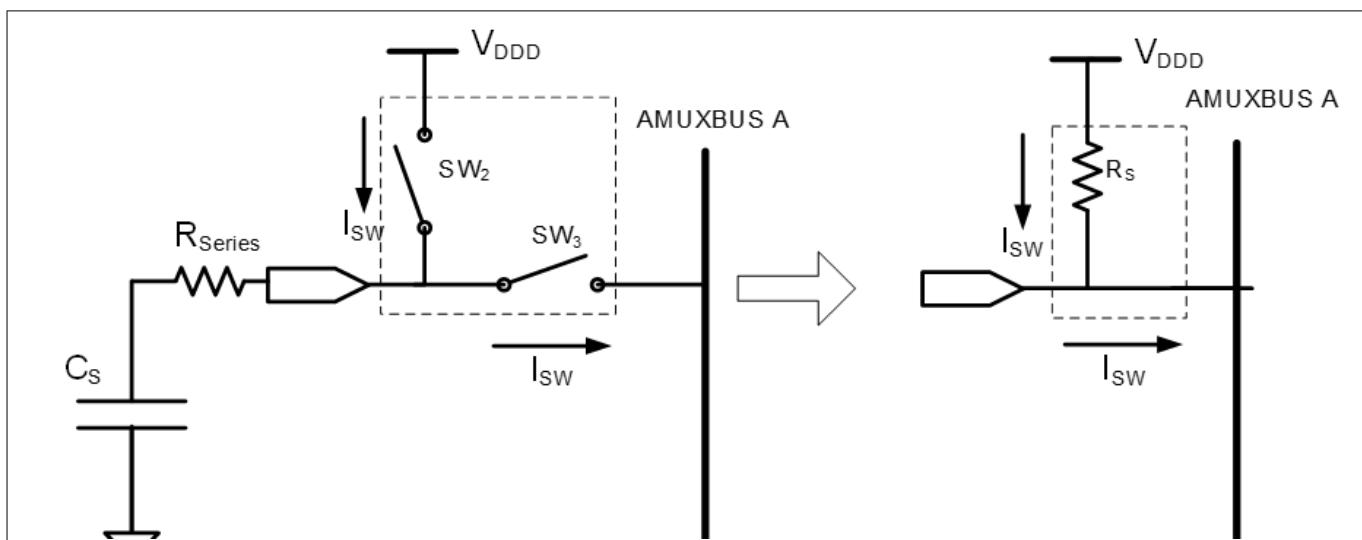


Figure 230 **GPIO cell sourcing current to AMUXBUS A**

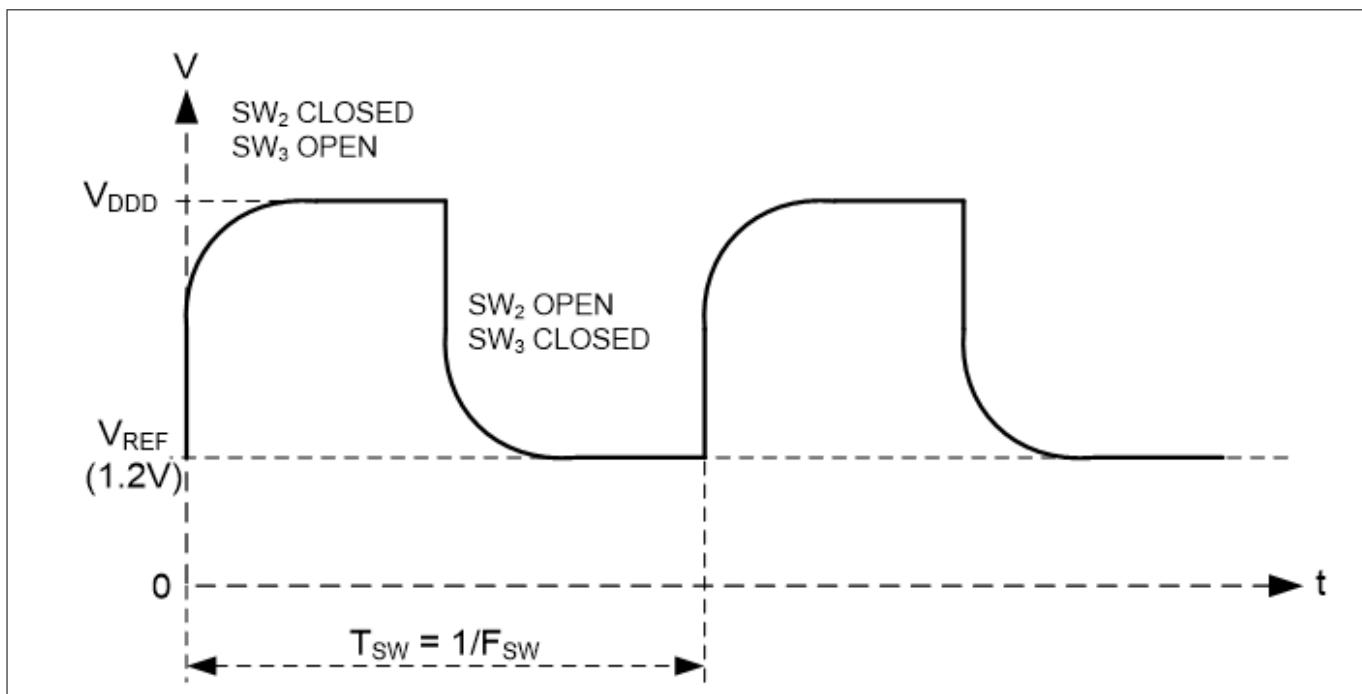


Figure 231 **Voltage across sensor capacitance**

~~5 PSoC™ 6 application notes~~

~~DRAFT~~
Equation 8 provides the value of average current supplied to AMUXBUS A.

$$I_{CS} = C_S F_{SW} (V_{DDD} - V_{REF})$$

Equation 8 Average current sourced to AMUXBUS A from GPIO through CAPSENSE™ sensor (ICS)

5.8.3.2.4 CAPSENSE™ clock generator

The CAPSENSE™ clock generator block generates the sense clock F_{SW} , and the modulation clock F_{MOD} , from the high-frequency system resource clock (HFCLK) or peripheral clock (PERI) depending on the PSoC™ device family as shown in [Figure 225](#).

Sense Clock

The sense clock, also referred to as the switching clock, drives the non-overlapping clocks to the GPIO cell switched capacitor circuits for the [GPIO cell capacitance to charge converter](#).

Sense clock can be sourced from three options: direct, 8-bit PRS, and 12-bit PRS. Some PSoC™ 4 and PSoC™ 6 MCU parts also support additional spread spectrum clock (SSCx) modes. For more details on the supported modes for PSoC™ device, see the [Component datasheet/middleware document](#).

Direct clock is a constant frequency sense clock source. When you chose this option, the sensor pin switches with a constant frequency clock with frequency as specified in the CAPSENSE™ component configuration window.

PRS clock implies that the sense clock is driven from a PRS block, which can generate either 8-bit or 12-bit PRS. Use of the PRS clock spreads the sense clock frequency over a wide frequency range by dividing the input clock using a PRS.

SSCx also spreads the sense clock frequency. It provides better noise immunity and reduces radiated electromagnetic emissions.

See [Manually tuning hardware parameters](#) for details on the clock source and frequency selection guidelines.

Modulator clock

The modulation clock is used by the [Sigma-delta converter](#). This clock determines the sensor scan time based on [Equation 8](#) and [Equation 9](#).

$$\text{Sensor scan time} = \text{Hardware scan time} + \text{Sensor Initialization time}$$

Equation 9 Sensor scan time

$$\text{Hardware scan time} = \frac{(2^{\text{Resolution}} - 1)}{\text{Modulator Clock Frequency}}$$

Equation 10 Hardware scan time

Where,

Resolution = [Scan resolution](#)

Sensor Initialization time = Time taken by the sensor to write to the internal registers and initiate a scan.

~~5 PSoC™ 6 application notes~~

~~DRAFT~~ 5.8.3.2.5 Sigma-delta converter

The sigma-delta converter converts the input current to a corresponding digital count. It consists of a sigma-delta converter and two current sourcing/sinking digital-to-analog converters (IDACs) called modulation IDAC and compensation IDAC as [Figure 225](#) shows.

The sigma-delta converter uses an external integrating capacitor, called modulator capacitor C_{MOD} , as [Figure 225](#) shows. Sigma-delta converter controls the modulation IDAC current by switching it ON or OFF corresponding to the small voltage variations across C_{MOD} to maintain the C_{MOD} voltage at V_{REF} . The recommended value of C_{MOD} is listed in [Table 68](#).

The sigma-delta converter can operate in either IDAC sourcing mode or IDAC sinking mode.

- **IDAC sourcing mode:** In this mode, the [GPIO cell capacitance to charge converter](#) sinks current from C_{MOD} through AMUXBUS A, and the IDACs then source current to AMUXBUS A to balance its voltage
- **IDAC sinking mode:** In this mode, the [GPIO cell capacitance to charge converter](#) sources current from C_{MOD} to AMUXBUS A and the IDACs sink current through AMUXBUS A to balance its voltage

In both the above-mentioned modes, the sigma delta converter can operate in either single IDAC mode or dual IDAC mode:

- In the single IDAC mode, the modulation IDAC is controlled by the sigma-delta converter; the compensation IDAC is always OFF
- In the dual IDAC mode, the modulation IDAC is controlled by the sigma-delta converter; the compensation IDAC is always ON

In the single IDAC mode, if ‘N’ is the resolution of the sigma-delta converter and I_{MOD} is the value of the modulation IDAC current, the approximate value of raw count in the IDAC Sourcing mode is given by [Equation 10](#).

$$\text{raw count} = \left(2^N - 1\right) \frac{V_{REF} F_{SW}}{I_{MOD}} C_S$$

Equation 11 Single IDAC sourcing raw count

Similarly, the approximate value of raw count in the IDAC sinking mode is given by [Equation 11](#).

$$\text{raw count} = \left(2^N - 1\right) \frac{(V_{DD} - V_{REF}) F_{SW}}{I_{MOD}} C_S$$

Equation 12 Single IDAC sinking raw count

In both cases, the raw count is proportional to sensor capacitance C_S . The raw count is then processed by the CAPSENSE™ CSD Component firmware to detect touches. The hardware parameters such as I_{MOD} , I_{COMP} , and F_{SW} , and the software parameters, should be tuned to optimum values for reliable touch detection. For an in-depth discussion of the tuning, see [CAPSENSE™ performance tuning](#).

In the dual IDAC mode, the compensation IDAC is always ON. If I_{COMP} is the compensation IDAC current, the equation for the raw count in the IDAC sourcing mode is given by [Equation 12](#).

$$\text{raw count} = \left(2^N - 1\right) \frac{V_{REF} F_{SW}}{I_{MOD}} C_S - \left(2^N - 1\right) \frac{I_{COMP}}{I_{MOD}}$$

Equation 13 Dual IDAC sourcing raw count

Raw count in the IDAC sinking mode is given by [Equation 13](#).

5 PSoC™ 6 application notes

$$\text{raw count} = \left(2^N - 1\right) \frac{V_{DD} - V_{REF} F_{SW}}{I_{MOD}} C_S - \left(2^N - 1\right) \frac{I_{COMP}}{I_{MOD}}$$

~~DRAFT~~ Equation 14 Dual IDAC sinking raw count

Note: Raw count values are always positive. It is thus imperative to ensure that I_{COMP} is less than $(V_{DD} - V_{REF})C_S F_{SW}$ for the IDAC sinking mode and I_{COMP} is less than $C_S F_{SW} V_{REF}$ for the IDAC Sourcing mode. [Equation 13](#) does not hold true if $I_{COMP} > V_{REF} C_S F_{SW}$ and [Equation 12](#) does not hold true if $I_{COMP} > (V_{DD} - V_{REF}) C_S F_{SW}$; in these cases, raw counts will be zero.

The relation between the parameters shown in the above equation to the CAPSENSE™ Component parameters is listed in [Table 36](#).

Table 36 Relationship between CAPSENSE™ raw count and CAPSENSE™ hardware parameters

Sl. No.	Parameter	Description	Comments
1	N	Scan resolution	Scan resolution is configurable from 6-bit to 16-bit. See Component datasheet/middleware document for details.
2	V_{REF}	N/A	The V_{REF} value is 1.2 V or configurable between 0.6 V to $V_{DDA} - 0.6$ V depending on the PSoC™ device family. See Component datasheet/middleware document for details.
3	F_{SW}	Sense clock frequency	Sense clock frequency and sense clock source decide the frequency at which the sensor is switching.
		Sense clock source	See Sense Clock for details.
4	I_{MOD}	Modulator IDAC	I_{MOD} = Modulation IDAC current
5	I_{COMP}	Compensation IDAC	I_{COMP} = Compensation IDAC current
6	V_{DD}	N/A	This parameter is the device supply voltage.
7	C_S	N/A	This parameter is the sensor parasitic capacitance.
8	N/A	Modulator clock frequency	Modulator clock divider does not impact raw count. See the Modulator clock section for more details.

5.8.3.2.6 Analog multiplexer (AMUX)

The sigma delta converter scans one sensor at a time. An analog multiplexer selects one of the GPIO cells and connects it to the input of the sigma delta converter, as [Figure 225](#) shows. The AMUXBUS A and the GPIO cell switches (see SW3 in [Figure 230](#)) forms this analog multiplexer. AMUXBUS A connects to all GPIOs that support CAPSENSE™. See the corresponding [Device datasheet](#) for a list of port pins that support CAPSENSE™. AMUXBUS A also connects the integrating capacitor C_{MOD} to the sigma-delta converter circuit. AMUXBUS B is used for shielding and is kept at V_{REF} when shield is enabled.

5.8.3.2.7 CAPSENSE™ CSD shielding

PSoC™ 4 and PSoC™ 6 MCU CAPSENSE™ supports shield electrodes for liquid tolerance and proximity sensing. CAPSENSE™ has a shielding circuit that drives the shield electrode with a replica of the sensor switching signal to nullify the potential difference between sensors and shield electrode. See [Driven-shield signal and shield](#)

~~5 PSoC™ 6 application notes~~

electrode Driven-shield signal and shield and [Effect of liquid droplets and liquid stream on a self-capacitance sensor](#) for details on how this is useful for liquid tolerance.

In the sensing circuit, the sigma delta converter keeps the AMUXBUS A at V_{REF} (see [Sigma-delta converter](#)). The GPIO cells generate the sensor waveforms by [switching the sensor](#) between AMUXBUS A and a supply rail (either V_{DD} or ground, depending on the configuration). The shielding circuit works in a similar way; AMUXBUS B is always kept at V_{REF} . The GPIO cell switches the shield between AMUXBUS B and a supply rail (either V_{DDD} or ground, the same configuration as the sensor). This process generates a replica of the sensor switching waveform on the shield electrode.

For a large shield layer with high parasitic capacitance, an external capacitor (Csh tank capacitor) is used to enhance the drive capacity of the shield electrode driver.

5.8.3.3 CAPSENSE™ CSX sensing method (third- and fourth- generation)

Figure 232 illustrates the CSX sensing circuit. The implementation uses the following hardware sub-blocks from CSD HW.

- An 8-bit IDAC and the sigma delta converter
- AMUXBUS A
- CAPSENSE™ clock generator for Tx clock and modulator clock
- V_{REF} and port pins for Tx and Rx electrodes and external caps
- Two external capacitors (CINTA and CINTB) (see [Table 68](#) for recommended value of these capacitors)

Note: *PSoC™ 4100 does not support the CSX sensing method.*

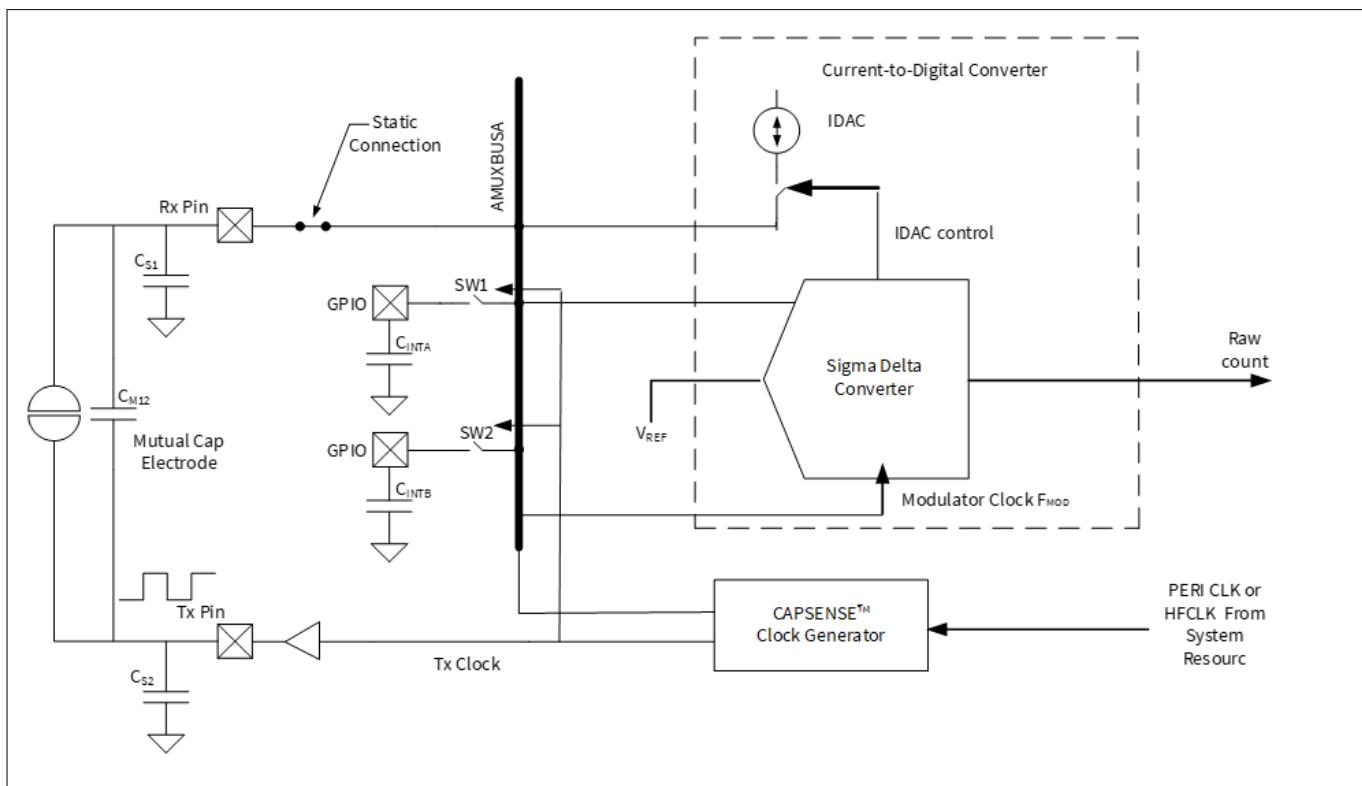


Figure 232 CAPSENSE™ CSX sensing method configuration

The CSX sensing method measures the mutual-capacitance between the Tx electrode and Rx electrode, as shown in [Figure 232](#). The Tx electrode is excited by a digital waveform (Tx clock), which switches between V_{DDIO} (or V_{DDD} if V_{DDIO} is not available in the given part number) and ground. The Rx electrode is statically connected

5 PSoC™ 6 application notes

~~DRAFT~~

to AMUXBUS A. The CSX method requires two external integration capacitors, C_{INTA} and C_{INTB} . The value of these capacitors is listed in [Table 68](#).

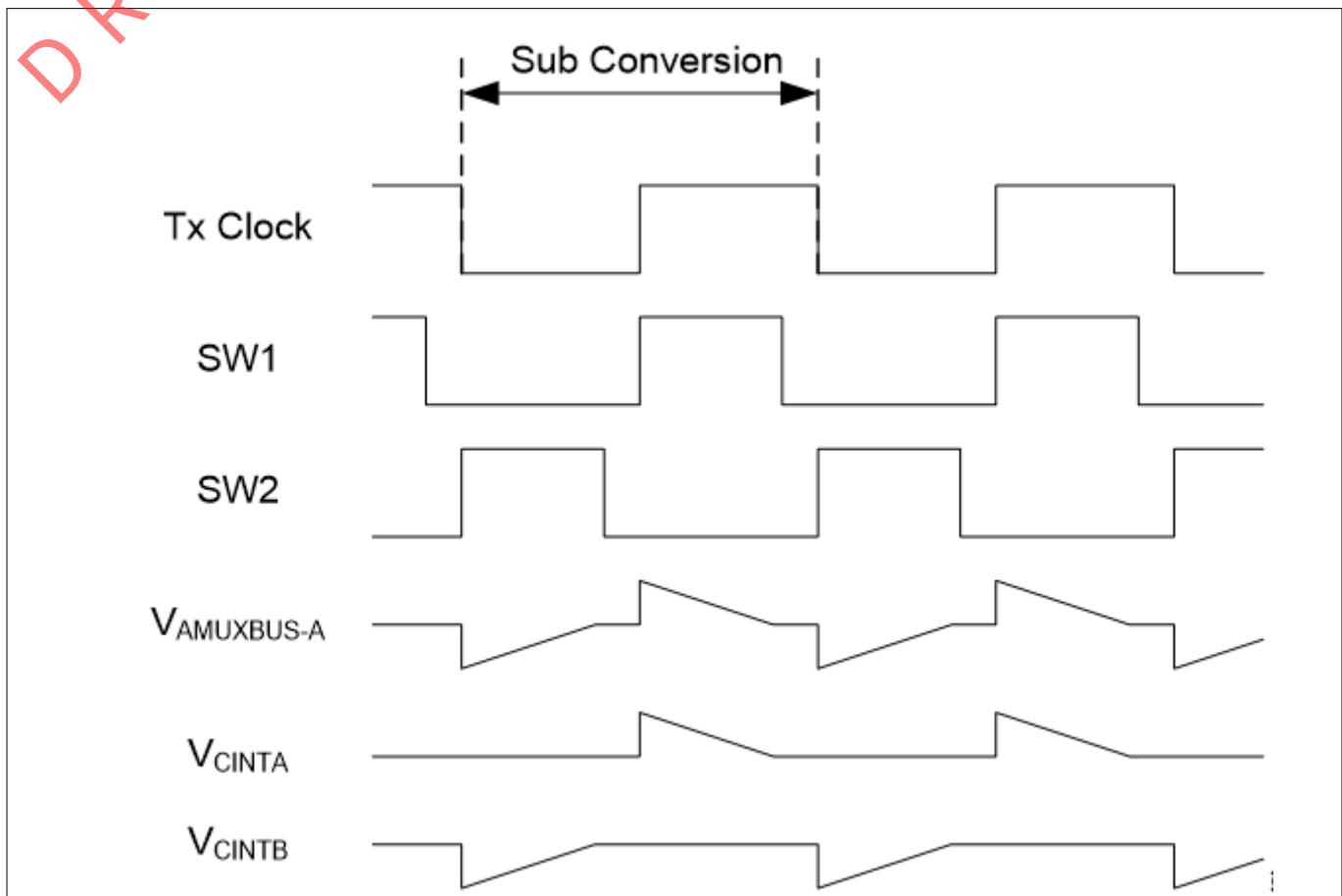


Figure 233 CSX sensing waveforms

[Figure 233](#) shows the voltage waveforms on the Tx electrode and C_{INTA} and C_{INTB} capacitors. The sampling – a process of producing a “sample” – is started by the firmware by initializing the voltage on both external capacitors to V_{REF} and performing a series of sub-conversions. A sub-conversion is a capacitance to count conversions performed within a Tx clock cycle. The sum of results of all sub-conversions in a sample is referred to as “raw count”.

During a sub-conversion, both SW1 and SW2 switches are operated in phase with the Tx clock. On the rising edge of the Tx clock, SW1 is closed (SW2 is open during this time) and charge flows from the Tx electrode to the Rx electrode. This charge is integrated onto the C_{INTA} capacitor, which increases the voltage on C_{INTA} . The IDAC is configured in sink mode to discharge the C_{INTA} capacitor back to voltage V_{REF} . On the falling edge of the Tx clock, SW2 is closed (SW1 is open during this time) and the charge flows from the Rx electrode to the Tx electrode. This causes the voltage on C_{INTB} to go below V_{REF} . The IDAC is configured in source mode to bring the voltage on C_{INTB} back to V_{REF} .

The charge transferred between Tx and Rx electrodes in both the cycles is proportional to mutual-capacitance, C_M , between the electrodes. The sigma delta converter controls IDAC for charging or discharging the external capacitors and also it measures the charging and discharging time in terms of modulator clock cycles for a sub-conversion. Multiple sub-conversions are performed during the CSX scanning and the result of each sub-conversion is accumulated to produce “raw count” for a sensor.

The modulator clock is used to measure the time taken to charge/discharge external capacitors within a Tx clock cycle. For this reason, modulator clock frequency must be always greater than Tx clock frequency; higher modulator clock frequency leads to better accuracy. For proper operation, the IDAC current should be set such that the C_{INTA} and C_{INTB} capacitors are charged/discharged within one Tx clock cycle. The CAPSENSE™

5 PSoC™ 6 application notes

~~DRAFT~~

Component/middleware provides an option to automatically calibrate the IDAC. It is recommended to enable this option.

$$\text{Rawcount}_{\text{Counter}} = \frac{2 V_{\text{TX}} F_{\text{TX}} C_M \text{MaxCount}}{\text{IDAC}}$$

$$\text{MaxCount} = \frac{F_{\text{Mod}} N_{\text{Sub}}}{F_{\text{TX}}}$$

Equation 15 Raw count relationship for mutual-capacitance sensing

Where,

IDAC = IDAC current

C_M = Mutual-capacitance between Tx and Rx electrodes

V_{TX} = Amplitude of the Tx signal

F_{TX} = Tx clock frequency

F_{Mod} = Modulator clock frequency

N_{Sub} = Number of sub-conversions

When you place a finger on the CSX button, the mutual-capacitance between Rx and Tx electrodes decreases, which decreases the raw count. This decrease in raw count from the hardware is inverted by the CAPSENSE™ Component to make it similar to the raw count change in CSD for a finger touch. The final resulting inverted raw count is given by [Equation 16](#).

$$\text{Rawcount}_{\text{Component}} = \text{MaxCount} - \text{Rawcount}_{\text{Counter}}$$

Equation 16 Formula to determine rawcount_{Component}

See [CSX sensing method \(third- and fourth-generation\)](#) for more details of CSX hardware parameters.

5.8.3.4 CAPSENSE™ CSD-RM sensing method (fifth-generation)

This section provides an overview of the CSD-RM architecture implemented in the Fifth-Generation CAPSENSE™ (known as multi sense converter (MSC)) devices. The main features include ratiometric sensing, differential mode of operation without the need of reference voltage, use of capacitor DACs (CDAC) in place of current DACs (IDAC) which improves noise performance.

5 PSoC™ 6 application notes

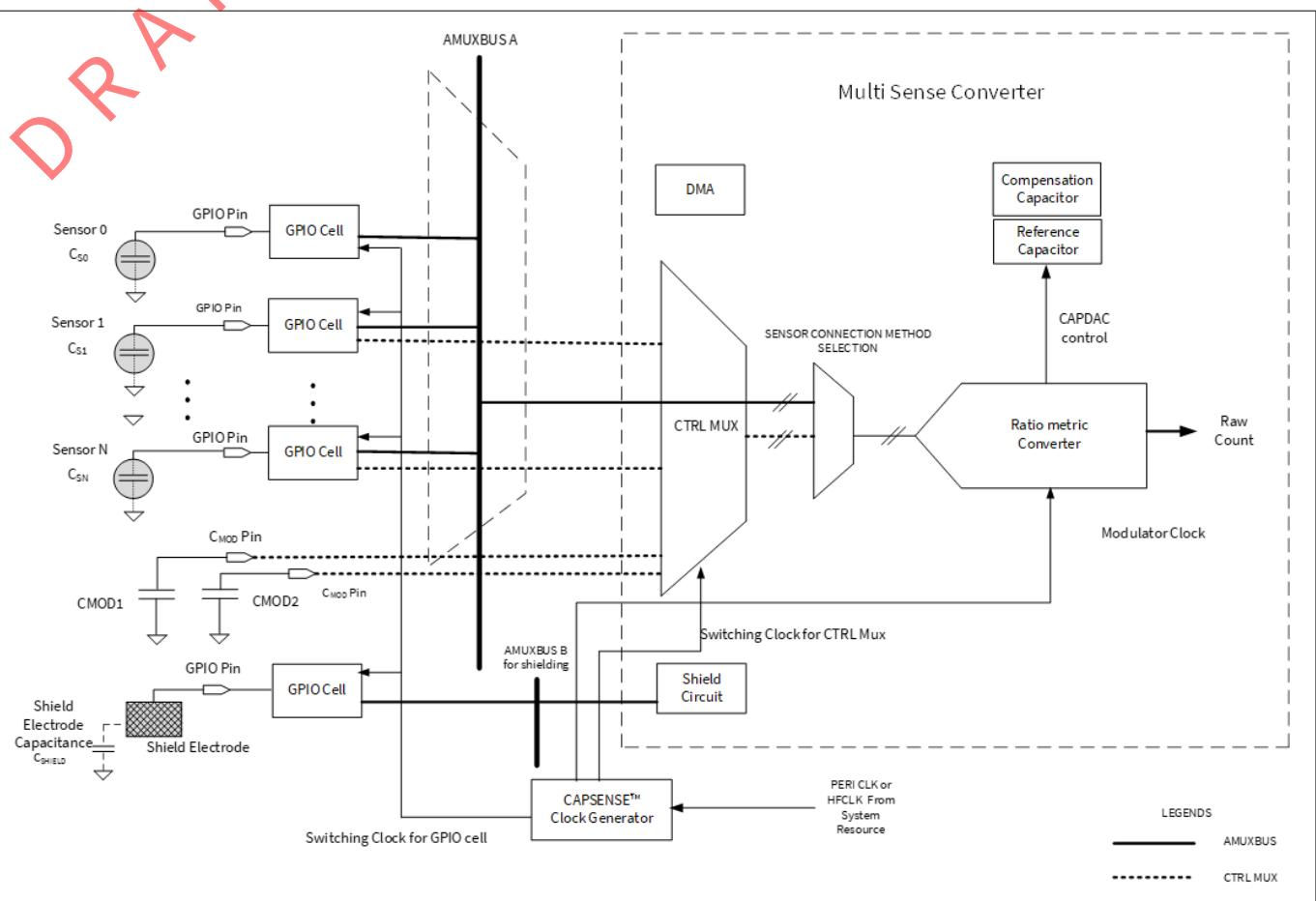


Figure 234 CAPSENSE™ CSD-RM (fifth-generation)

5.8.3.4.1 GPIO cell capacitance to charge converter

Chapter 5.8.3.2.1 explains the GPIO cell configuration. In the Fifth-Generation architecture, the sensor is either interfaced to the AMUX (as before) or a new control MUX matrix which supports autonomous scanning (limited number of pins supported). The GPIO cells are configured as switched-capacitance circuits that convert sensor capacitances into equivalent charge transfer. Figure 235 shows the GPIO cell structure.

5 PSoC™ 6 application notes

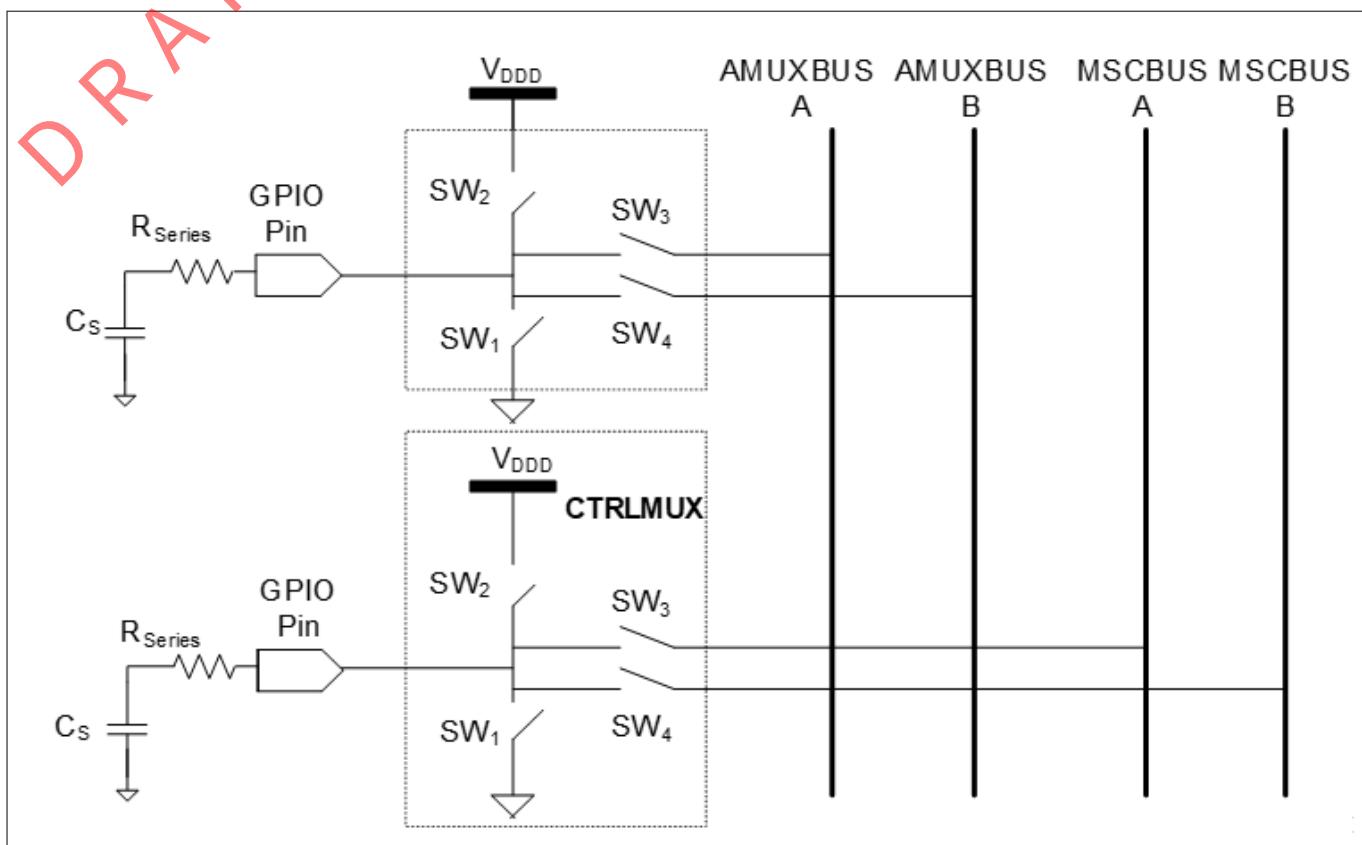


Figure 235 GPIO cell structure

Four non-overlapping, out-of-phase clocks of frequency F_{SW} control the switches (SW_1 , SW_2 , SW_3 and SW_4) as Figure 236 shows.

5 PSoC™ 6 application notes

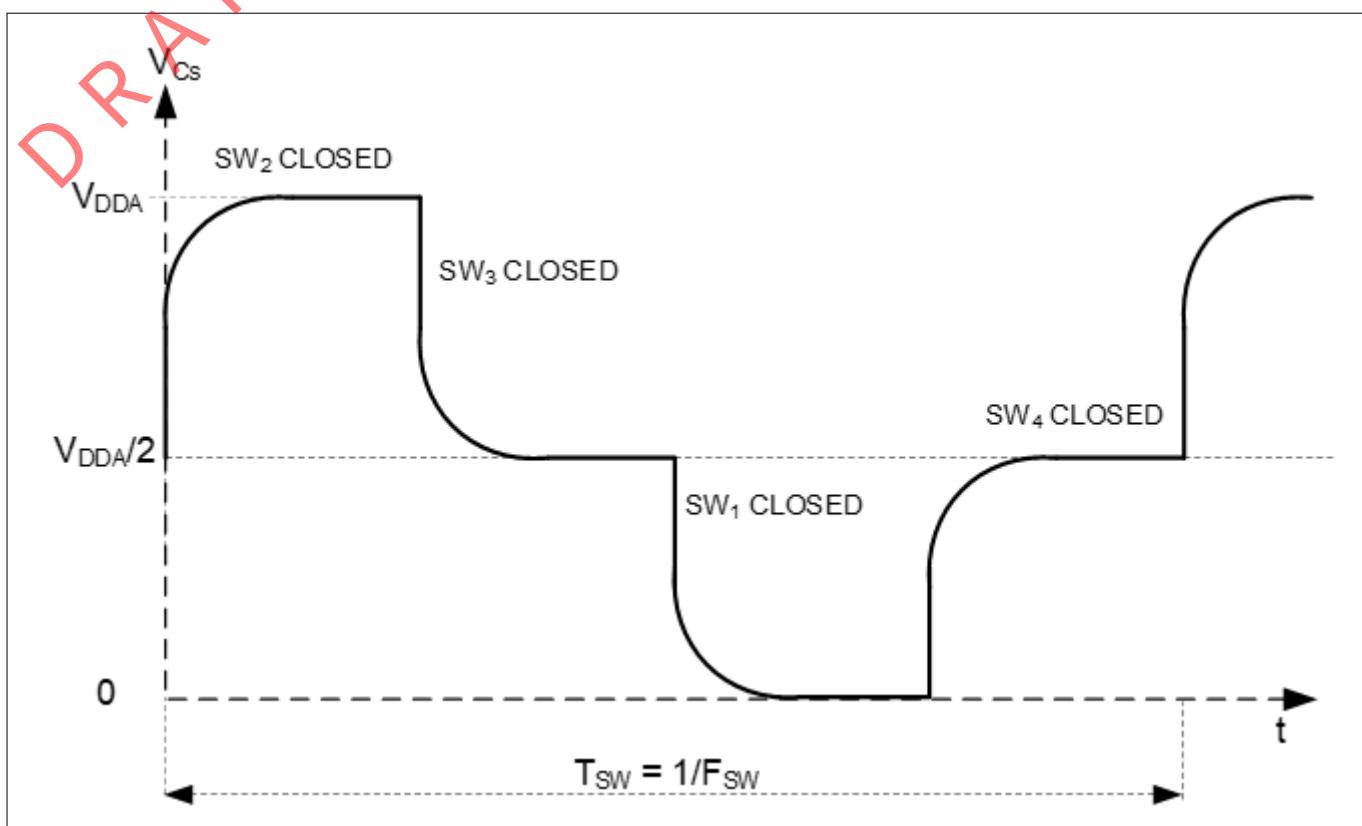


Figure 236 Voltage across sensor capacitance

5.8.3.4.2 Capacitor DACs (CDACs)

IDACs are replaced by CDACs in the Fifth-Generation CAPSENSE™ architecture. It consists of two CDACs, a reference capacitor DAC and a compensation capacitor DAC. In each sense clock period the sensor capacitance, as mentioned in [GPIO cell capacitance to charge converter](#), transfers charge to both C_{MOD} in a way that it unbalances the voltage between the C_{MOD} 's. Both capacitor DACs are switched onto C_{MOD} multiple times during a sense clock period to balance the C_{MOD} 's back to their original voltage. Number of cycles required by the reference capacitor DAC to balance is proportional to the self-capacitance of the sensors.

5.8.3.4.3 CAPSENSE™ clock generator

This block generates the sense clock F_{SW} , and the modulation clock F_{MOD} , from the high-frequency system resource clock (HFCLK) or peripheral clock (PERI) depending on the PSoC™ device family.

Sense clock

CAPSENSE™ clock generation is similar to that in the older generation as explained in [Chapter](#).

Modulator clock

The modulation clock is used by the [Ratiometric sensing technology](#). This clock determines the sensor scan time based on [Equation 16](#) and [Equation 17](#).

$$\text{Sensor scan time} = \text{Hardware scan time} + \text{Sensor initialization time}$$

Equation 17 **Sensor scan time**

5 PSoC™ 6 application notes

~~DRAFT~~

$$\text{Hardware scan time} = \frac{(\text{Number of subconversions})}{\text{Sense clock frequency}}$$

Equation 18 Hardware scan time

Where,

Number of subconversions = Total number or sub-conversions in single scan

Subconversion = Capacitance to count conversions performed within a sense clock cycle

Sensor initialization time = Time taken by the sensor to write to the internal registers and initiate a scan

5.8.3.4.4 Ratiometric sensing technology

It consists of a ratiometric converter and two CDACs, a reference capacitor DAC and a compensation capacitor DAC. In each sense clock period the sensor capacitance, as mentioned in [GPIO cell capacitance to charge converter](#), transfers charge to both C_{MOD} in a way that it unbalances the voltage between the C_{MOD} 's. The Ratiometric converter controls the reference CDAC by switching it ON or OFF corresponding to the small voltage variations across two C_{MOD} 's to maintain the C_{MOD} 's voltage at same level. Number of cycles required by the reference capacitor DAC to balance the voltage between the C_{MOD} 's is proportional to the self-capacitance of the sensors.

The compensation capacitor is used to compensate excess mutual-capacitance from the sensor to increase the sensitivity. The number of times it is switched depends on the amount of charge the user application is trying to compensate (remove) from the sensor mutual-capacitance.

The ratiometric converter can operate in either single CDAC mode or dual CDAC mode.

- In the single CDAC mode, the reference CDAC is controlled by the ratiometric converter; the compensation CDAC is always OFF
- In the dual CDAC mode, the reference CDAC is controlled by the ratiometric converter; the compensation CDAC is always ON. Reference CDAC is capable of compensating up to 95%, results in increased signal as explained in [Conversion gain and CAPSENSE signal](#)

In the single CDAC mode, if C_{ref} is the value of the reference CDAC, the approximate value of raw count is given by [Equation 18](#).

$$\text{Rawcount} = \text{Maxcount} \cdot \frac{C_S}{SnsClk_{Div} \cdot C_{ref}}$$

Equation 19 CSD-RM single CDAC raw count

In the dual CDAC mode, the compensation CDAC is always ON. If C_{comp} is the compensation CDAC, the equation for the raw count is given by [Equation 19](#).

$$\text{Rawcount} = \text{Maxcount} \cdot \frac{C_S - 2 \cdot \frac{SnsClk_{Div}}{CompClk_{Div}} \cdot C_{comp}}{SnsClk_{Div} \cdot C_{ref}}$$

Equation 20 CSD-RM dual CDAC raw count

Where,

$\text{MaxCount} = N_{Sub} \cdot SnsClk_{Div}$

N_{Sub} = Number of sub-conversions

$SnsClk_{Div}$ = Sense clock divider

$CompClk_{Div}$ = Compensation CDAC divider

5 PSoC™ 6 application notes

~~DO NOT USE~~
 C_S = Sensor capacitance

C_{ref} = Reference capacitance

C_{comp} = Compensation capacitance

As per [Equation 18](#), the output raw count is proportional to the ratio of sensor capacitance to the reference capacitance, and hence the name Ratiometric Sensing.

Noise improvement is one of the main advantages of Fifth-Generation over previous generation of CAPSENSE™ technology. The dominant noise sources in the Fourth-Generation are current (I_{MOD}), reference voltage (V_{REF}), clock jitter (F_{SW}) (see [Equation 13](#)). These noise sources have been removed for the Fifth-Generation (see [Equation 19](#)). The IDAC has been replaced with CDAC. The system has been made fully differential, so it does not need V_{REF} . The CAPSENSE™ architecture is no longer affected by jitter as the scan result is now based on the edges of the clock rather than the duration of the clock.

5.8.3.4.5 Analog multiplexer (AMUX) and control matrix (CTRLMUX)

Another feature introduced in the Fifth-Generation is the control matrix (CTRLMUX) as shown in [Figure 234](#). The CTRLMUX enables autonomous scanning and provides immunity to on-chip IO noise. The CTRLMUX allows the CAPSENSE™ IP to directly handle the sensor inputs¹⁶ (in addition to the traditional GPIO mode), and hence supports autonomous scanning of the sensors without the CPU.

5.8.3.4.6 CAPSENSE™ CSDRM shielding

PSoC™ 4 CAPSENSE™ supports shield electrodes for liquid tolerance and proximity sensing. The purpose of the shielding is to remove the parasitic capacitance between sensor and shield electrodes. See [Driven-shield signal and shield electrode](#) and [Effect of liquid droplets and liquid stream on a self-capacitance sensor](#) for details on how this is useful for liquid tolerance. The Fifth-Generation CAPSENSE™ architecture supports two shield modes – active and passive shielding.

Active shielding

In active shielding mode, shield circuit drives the shield electrode with a replica of the sensor signal using a buffer as shown in [Figure 237](#). This nullify the potential difference between sensors and shield electrode.

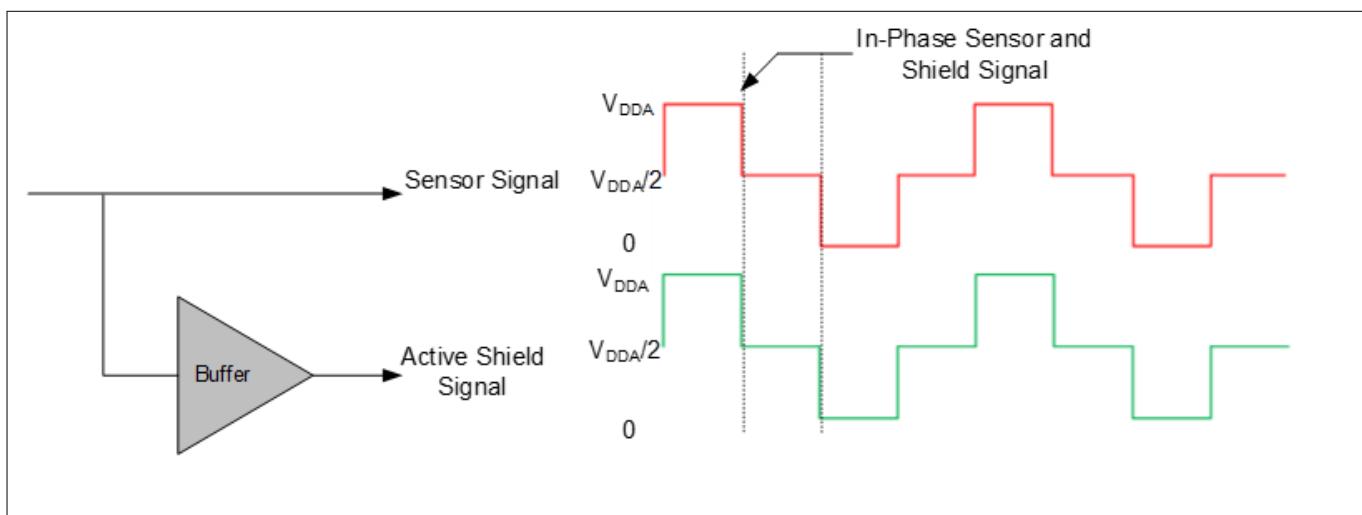


Figure 237 **Active shield signal**

¹⁶ Supports limited number of inputs. Refer to the [Device](#) for more details.

~~5 PSoC™ 6 application notes~~

~~Passive shielding~~

In passive shielding mode there is no buffer used, instead shield is switched between V_{DDA} and GND as shown in [Figure 238](#). The switching is controlled in such a way that the net charge between sensor and shield is nullified every two sense clocks.

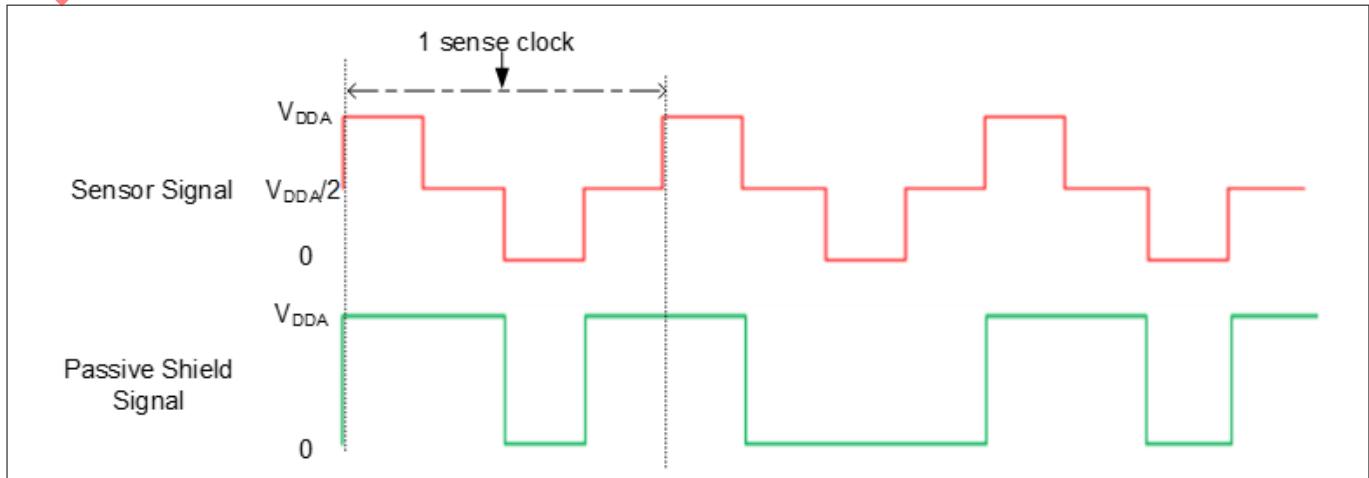


Figure 238 Passive shield signal

[Table 37](#) provides the comparison of features of active shielding vs passive shielding features.

Table 37 Active vs passive shielding

Feature	Active shielding	Passive shielding	Effect
Performance	Higher	Lower	Active shielding is preferred for high performance applications.
Power impact	Higher	Lower	Passive shielding is preferred for low power applications.

5.8.3.5 CAPSENSE™ CSX-RM sensing method (fifth-generation)

[Figure 239](#) illustrates the CSX-RM sensing circuit. The implementation uses the following hardware subblocks:

- Two 8-bit capacitor DACs and ratiometric converter
- AMUXBUS and CTRLMUX
- CAPSENSE™ clock generator for Tx clock and modulator clock
- Port pins for Tx and Rx electrodes and external caps
- Two external capacitors (C_{MOD1} and C_{MOD2})

5 PSoC™ 6 application notes

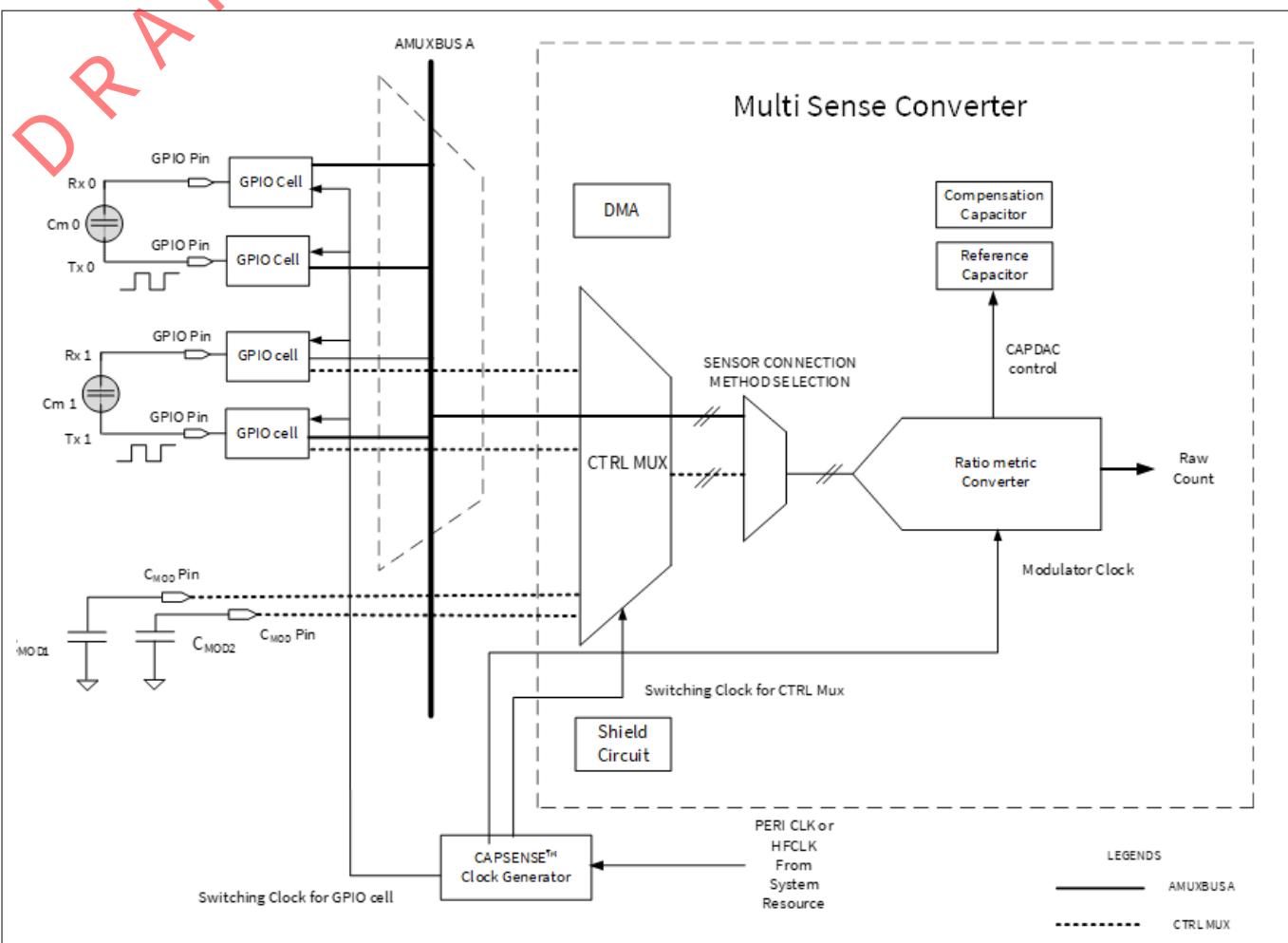


Figure 239 CAPSENSE™ CSX-RM sensing method configuration

The CSX-RM sensing method measures the mutual-capacitance between the Tx electrode and Rx electrode, as shown in [Figure 239](#). The Tx electrode is activated by a digital waveform (Tx clock), which switches between V_{DDA} and ground. The Rx electrode is statically connected to AMUXBUS A or CTRLMUX. The CSX-RM method requires two external integration capacitors, C_{MOD1} and C_{MOD2} .

The sampling – a process of producing a “sample” – is started by the firmware by initializing the voltage on both external capacitors (C_{MOD}) to $V_{DDA}/2$ and performing a series of sub-conversions. A sub-conversion is a capacitance to count conversions performed within a Tx clock cycle. The sum of results of all sub-conversions in a sample is referred to as “raw count”.

On the rising and falling edge of the Tx clock, charge flows from the Tx electrode to the Rx electrode. In such a way that it unbalances the voltage between the external C_{MOD} capacitors. Both capacitor DACs (reference and compensation capacitor DACs) are switched onto C_{MOD} multiple times during a sense clock period to balance the C_{MOD} ’s back to their original voltage. Number of cycles required by the reference capacitor DAC to balance is proportional to the mutual-capacitance, C_m , between the electrodes.

The number of times the reference capacitor is switched with respect to the modulator clock is denoted by the Tx clock divider value according to [Equation 21](#).

$$TxClk_{Div} = \frac{F_{Mod}}{F_{Tx}}$$

Equation 21 Tx clock divider

Where,

5 PSoC™ 6 application notes

~~DRAFT~~
 $TxClk_{Div}$ = Tx clock divider

F_{Mod} = Modulator frequency

F_{Tx} = Tx clock frequency

The compensation capacitor is used to compensate excess mutual-capacitance from the sensor to increase the sensitivity. The number of times it is switched depends on the amount of charge the user application is trying to compensate (remove) from the sensor mutual-capacitance. The number of times the compensation capacitor is switched with respect to the modulator clock is denoted by the value of the compensation CDAC divider according to the [Equation 21](#). The CDAC compensation clock divider must be less than or equal to the Tx clock divider.

$$CompClk_{Div} = \frac{F_{MOD}}{F_{comp}}$$

Equation 22 Compensation CDAC divider

Where,

$CompClk_{Div}$ = Compensation CDAC divider

$F_{M\ MOD}$ = Modulator frequency

F_{comp} = Compensation CDAC clock frequency

5.8.3.5.1 Ratiometric sensing technology

The ratiometric converter gives an equivalent raw count which is proportional to the sensor mutual-capacitance after each scan. The ratiometric converter can operate in either single CDAC mode or dual CDAC mode.

- In the single CDAC mode, the reference CDAC is controlled by the ratiometric converter; the compensation CDAC is always OFF
- In the dual CDAC mode, the reference CDAC is controlled by the ratiometric converter; the compensation CDAC is always ON. Compensation CDAC is capable of compensating up to 95%, results in increased signal as explained in [Conversion gain and CAPSENSE signal](#)

In the single CDAC mode, if C_{ref} is the value of the reference CDAC, the approximate value of raw count is given by [Equation 22](#).

$$\text{Rawcount} = \text{Maxcount} \cdot \frac{C_M}{TxClk_{Div} \cdot \left(\frac{C_{ref}}{2} \right)}$$

Equation 23 CSX-RM single CDAC raw count

In the dual CDAC mode, the compensation CDAC is always ON. If C_{comp} is the compensation CDAC, the equation for the raw count is given by [Equation 23](#).

$$\text{Rawcount} = \text{Maxcount} \cdot \frac{C_M - \frac{TxClk_{Div}}{CompClk_{Div}} \cdot C_{comp}}{TxClk_{Div} \cdot \left(\frac{C_{ref}}{2} \right)}$$

Equation 24 CSX-RM dual CDAC raw count

Where,

$\text{MaxCount} = N_{Sub} \cdot TxClk_{Div}$

N_{Sub} = Number of sub-conversions

5 PSoC™ 6 application notes

~~DO NOT USE~~
TxClk_{Div} = Tx clock divider

CompClk_{Div} = CDAC compensation divider

C_M = Mutual-capacitance of the sensor

C_{ref} = Reference capacitance

C_{comp} = Compensation capacitance

According to [Equation 23](#), the output raw count is proportional to the ratio of mutual-capacitance of the sensor to the reference capacitance, and hence the name ratiometric sensing.

5.8.3.6 Autonomous scanning

In previous generation CAPSENSE™ technology, after each scan, CPU is interrupted to configure next sensor. Autonomous scanning mode in Fifth-Generation CAPSENSE™ technology avoids the CPU intervention for scanning every next sensor. This significantly reduces the CPU bandwidth required for scanning widgets with large number of sensors. Autonomous scanning requires features such as CTRLMUX and DMA. As the number of pins supported with CTRLMUX is limited, number of pins supporting autonomous scanning is also limited. See [Configuring autonomous scan](#) section for more details.

5.8.3.7 Usage of multiple channels

The PSoC™ 4100S Max device supports two Fifth-Generation CAPSENSE™ Blocks – MSC0 and MSC1. Each block has the same functionality and performance as explained in the [CAPSENSE™ CSD-RM sensing method \(fifth-generation\)](#) and [CAPSENSE™ CSX-RM sensing method \(fifth-generation\)](#) sections. Each instance can be considered as a channel and multiple instances imply multiple channels. Multi-channel behavior can be supported by multiple instances in single chip and/or having multiple chips. The operation of the channels is synchronized and operate in lockstep when scanning the sensors hooked in to the channels. Lockstep guarantees clock synchronization and avoid any cross-channel noise due to un-synchronized sense clocks. See [Multi-channel scanning](#) section for more details.

~~5 PSoC™ 6 application notes~~

~~DRAFT~~ 5.8.4 CAPSENSE™ design and development tools

This chapter introduces the available software tools, such as PSoC™ Creator and ModusToolbox™, to develop your CAPSENSE™ application. For more details, see the user manual of the respective IDE. [Table 38](#) shows the supported devices and the CAPSENSE™ component/middleware version in PSoC™ Creator and ModusToolbox™.

Table 38 Tools and supported devices

Devices	Software tool	CAPSENSE™ library
PSoC™ 4000S, PSoC™ 4100S, PSoC™ 4100S Plus, PSoC™ 4100S Plus 256K, PSoC™ 4500S	ModusToolbox™, PSoC™ Creator	CAPSENSE™ middleware, CAPSENSE™ component
PSoC™ 4100S Max, All PSoC™ 6 devices	ModusToolbox™	CAPSENSE™ middleware
All other PSoC™ 4 devices	PSoC™ Creator	CAPSENSE™ component

5.8.4.1 PSoC™ Creator

PSoC™ Creator is a state-of-the-art, easy-to-use IDE. It offers a unique combination of hardware configuration and software development based on classical schematic entry. You can develop applications in a drag-and-drop design environment using a library of Components. For details, see the [PSoC™ Creator home page](#).

5.8.4.1.1 CAPSENSE™ component

PSoC™ Creator provides a CAPSENSE™ component, which is used to create a capacitive touch system in PSoC™ by simply configuring this Component. The CAPSENSE™ component also provides an application programming interface (API) to simplify firmware development. Some PSoC™ 4 Bluetooth® LE and PSoC™ 6 MCU devices also support a CAPSENSE™ Gesture Component (see the corresponding [Device datasheet](#) to see if your device supports this Component).

5 PSoC™ 6 application notes

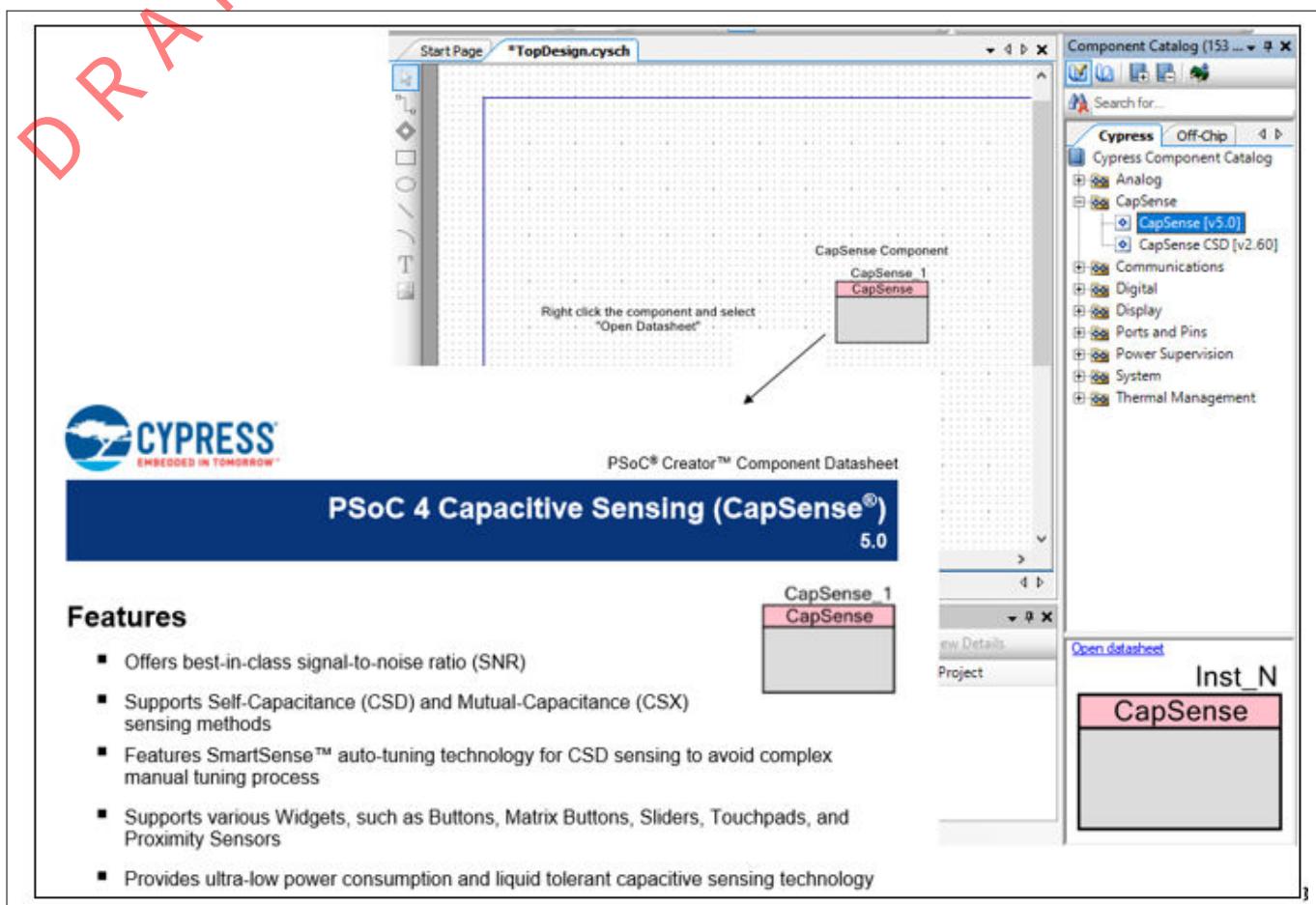


Figure 240 PSoC™ Creator component placement

Each CAPSENSE™ component has an associated datasheet that explains details about the Component. To open the Component datasheet, right-click the Component and select **Open Datasheet**.

The CAPSENSE™ component also has a Tuner GUI, called the [Tuner GUI](#), to help with the tuning process.

5.8.4.1.2 CapSense_ADC component

The CapSense_ADC¹⁷⁾ component is only applicable for the PSoC™ 4S-Series, PSoC™ 4100S Plus, PSoC™ 4100PS, and PSoC™ 6 MCU devices. This component should be used when both CAPSENSE™ and ADC operations are required. This component allows using the CAPSENSE™ block for ADC operation and touch functionality in a time-multiplexed manner.

5.8.4.1.3 Tuner GUI

Tuner helper is included with the CAPSENSE™ component and assists in tuning CAPSENSE™ parameters and monitoring sensor data such as raw count, baseline, and difference count. Refer the [Component datasheet/middleware document](#) for the detailed procedure on how to use Tuner GUI.

5.8.4.1.4 Example projects

You can use the CAPSENSE™ example projects provided in PSoC™ Creator to learn schematic entry and firmware development. To find a CAPSENSE™ example project, go to the PSoC™ Creator start Page, click **Find Code**

¹⁷ CapSense_ADC is not supported in devices with Fifth-Generation CAPSENSE™ block.

5 PSoC™ 6 application notes

~~DRAFT~~
Example ..., and select the appropriate architecture, as Figure 241 shows. You can also filter for a project by writing partial or complete project name in the **Filter by** field.

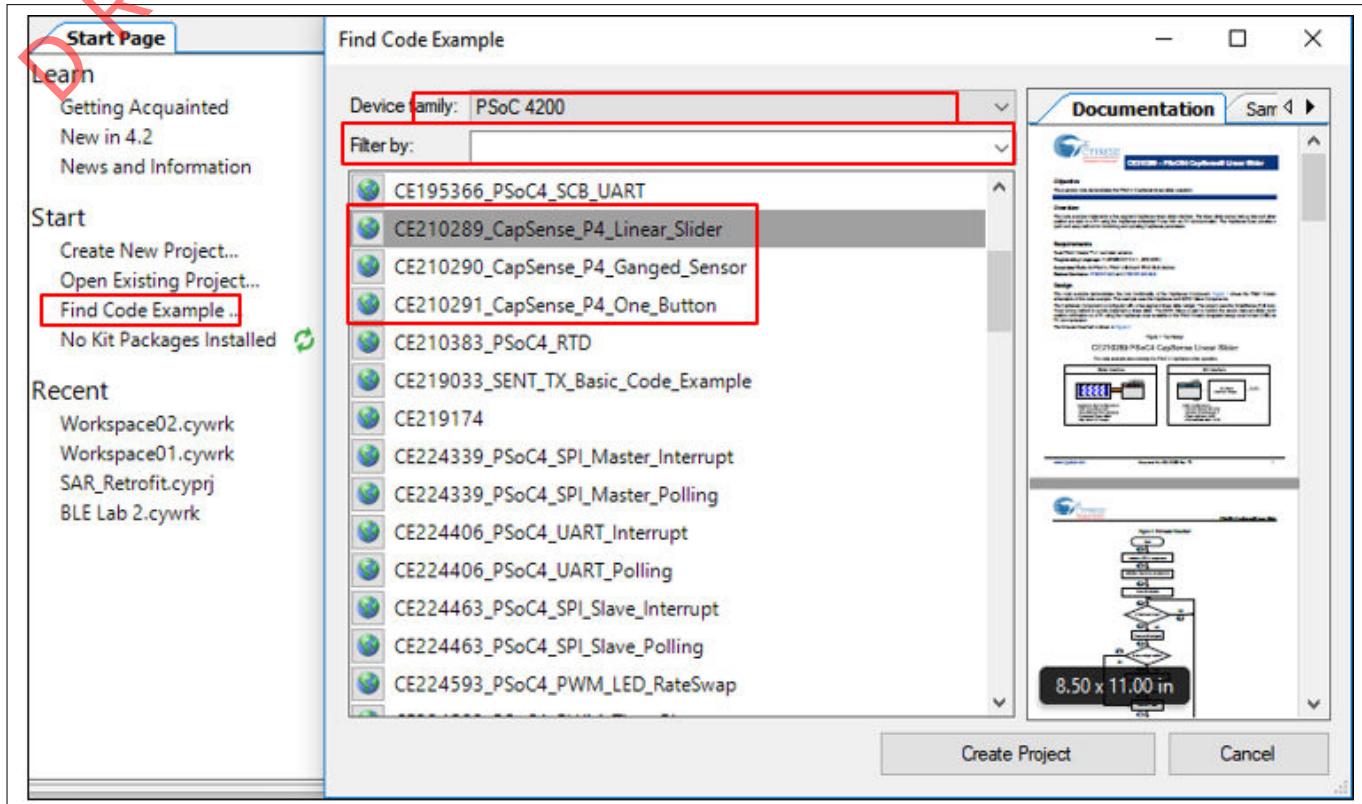


Figure 241 PSoC™ Creator example project

5.8.4.2 ModusToolbox™

ModusToolbox™ software suite is used for the development of PSoC™ 6 and PSoC™ 4¹⁸ based CAPSENSE™ applications. You can download ModusToolbox™ from [here](#). Before you start working with this software, It is recommended that you go through the [Quick start guide](#) and [user guide](#). If you have ModusToolbox™ IDE installed in your system, you can create a CAPSENSE™ application for the devices supported in ModusToolbox™.

5.8.4.2.1 CAPSENSE™ middleware

ModusToolbox™ provides a CAPSENSE™ middleware, which can be used to create a capacitive touch system in PSoC™ by simply configuring parameters in the CAPSENSE™ configuration tool. The middleware also provides an application programming interface (APIs) to simplify firmware development. See the [CAPSENSE™ middleware library](#) for more details.

5.8.4.2.2 CAPSENSE™ configurator

The CAPSENSE™ configurator tool in ModusToolbox™ is similar to that in PSoC™ Creator which is used to configure the CAPSENSE™ hardware and software parameters. For more details on configuring CAPSENSE™ in ModusToolbox™, see the [ModusToolbox™ CAPSENSE™ configurator guide](#) and [CAPSENSE™ middleware library](#). Figure 242 shows how to open the CAPSENSE™ configuration tool in ModusToolbox™. Alternatively, it can also be opened from the **Quick panel** in the ModusToolbox™. For simplicity of documentation, this design guide shows selecting the CAPSENSE™ parameter in PSoC™ Creator CAPSENSE™ component.

¹⁸ See [Table 38](#) for supported PSoC™ 4 devices in ModusToolbox™.

5 PSoC™ 6 application notes

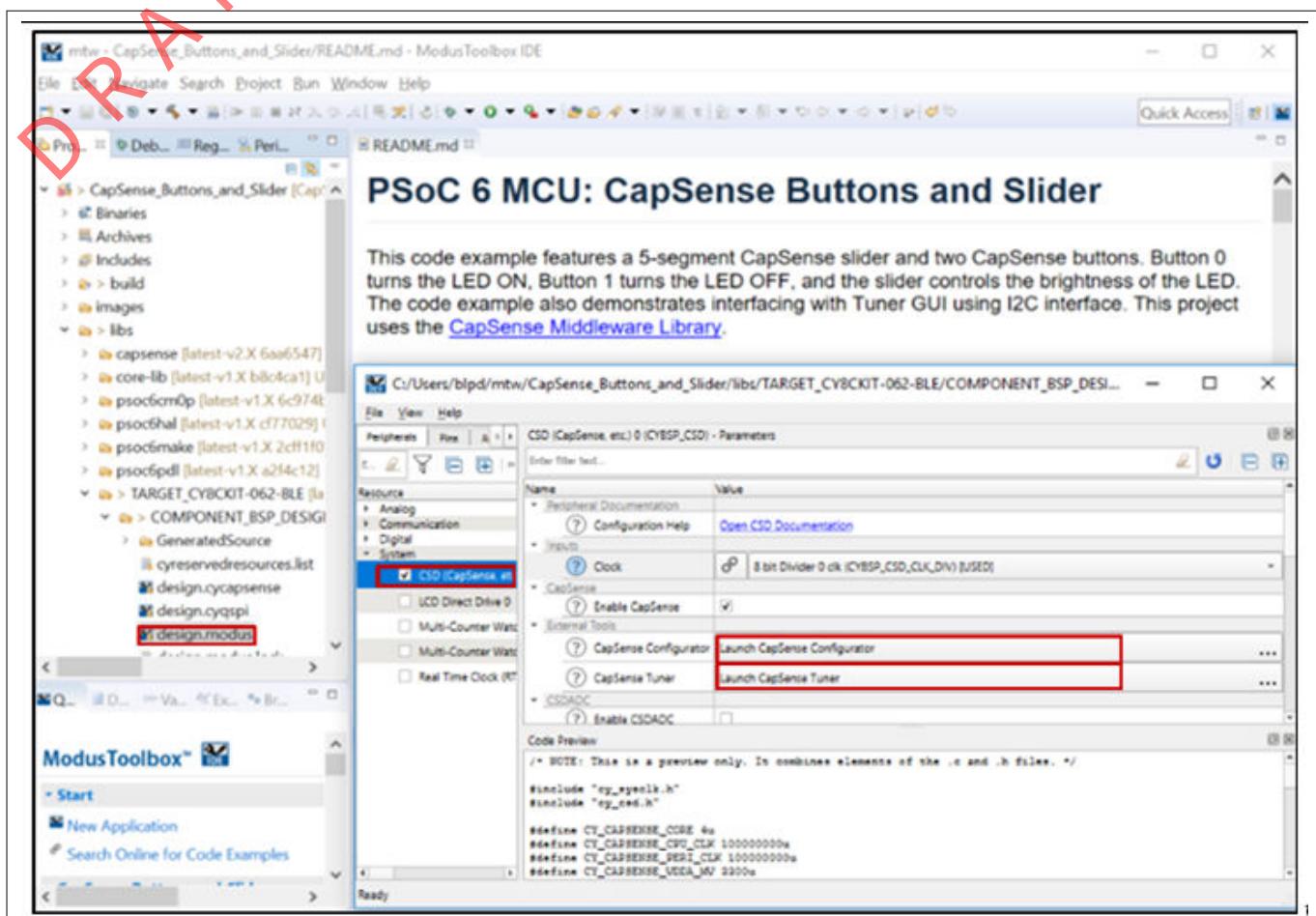


Figure 242 CAPSENSE™ configurator tool in ModusToolbox™

5.8.4.2.3 CSDADC middleware

This CSDADC middleware¹⁹ should be used when both the CAPSENSE™ and ADC operations are required. This middleware allows using the CAPSENSE™ hardware block for ADC operation and touch functionality in a time-multiplexed manner. It could be used for all three sensing modes that is, CSD, ADC, and CSX. See the [CSDADC middleware library](#) documentation for more details.

5.8.4.2.4 CSDIDAC middleware

The CSDIDAC middleware allows you to use the CAPSENSE™ IDAC in a standalone mode. You can use this middleware if you are not using CAPSENSE™ middleware or if you are using only one IDAC for CAPSENSE™. See the [CSDADC middleware library](#) documentation.

5.8.4.2.5 CAPSENSE™ Tuner

ModusToolbox™ also supports a GUI tool that can be used for tuning CAPSENSE™ parameters. This tool can be opened from the [Device configurator](#) by selecting Launch CapSense Tuner as shown in Figure 242. See the [CAPSENSE™ tuner guide](#) documentation.

¹⁹ CapSense_ADC is not supported in devices with Fifth-Generation CAPSENSE™ block.

~~5 PSoC™ 6 application notes~~

~~5.8.4.2.6 Example projects~~

To quickly start the CAPSENSE™ system design, start with the example projects provided in ModusToolbox™. You can find a CAPSENSE™ example project by navigating to **File > New > ModusToolbox Application**. Choose the appropriate Board Support Package with a device. **Figure 243** shows creating a CAPSENSE™ CSD Button example starter code in ModusToolbox™ from the list of available code examples.

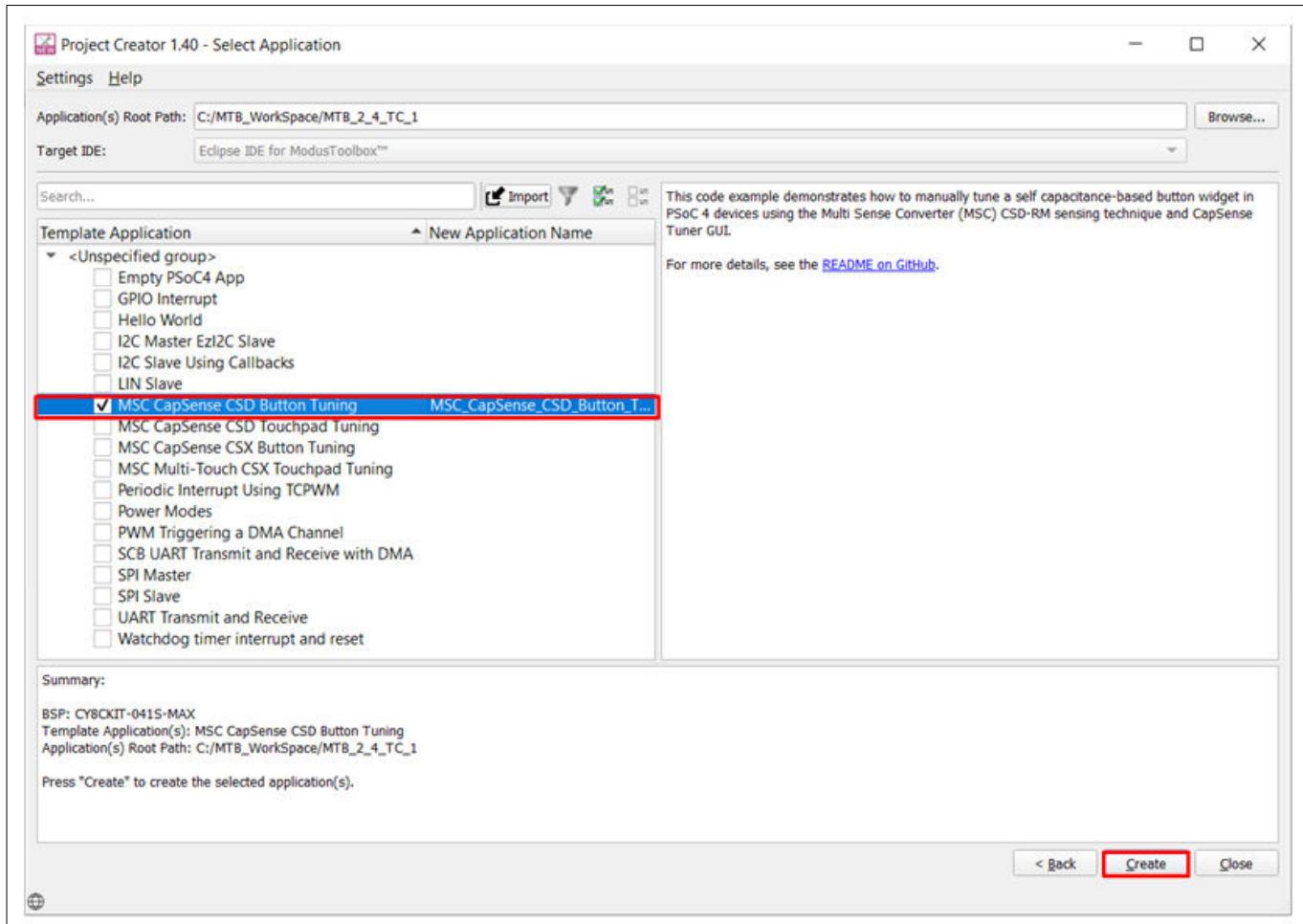


Figure 243 Creating CAPSENSE™ CSD Button example project in ModusToolbox™

5.8.4.3 Hardware kits

Table 39 lists the development kits that support evaluation of PSoC™ 4 and PSoC™ 6 CAPSENSE™.

Table 39 PSoC™ 4 and PSoC™ 6 CAPSENSE™ development kits

Development kit	Supported CAPSENSE™ features
PSoC™ 4000 pioneer kit (CY8CKIT-040)	A 5x6 CAPSENSE™ touchpad and a wire proximity sensor
PSoC™ 4 S-series pioneer kit (CY8CKIT-041)	Two self- or mutual-capacitive sensing buttons A 7x7 self- or mutual-capacitive sensing touchpad
PSoC™ 4 S-series prototyping kit (CY8CKIT-145)	Three self- or mutual-capacitive sensing buttons A five-segment self- or mutual-capacitive sensing linear slider

(table continues...)

~~DRAFT~~
5 PSoC™ 6 application notes

Table 39 (continued) PSoC™ 4 and PSoC™ 6 CAPSENSE™ development kits

Development kit	Supported CAPSENSE™ features
PSoC™ 4100S Plus prototyping kit (CY8CKIT-149)	Three self- or mutual-capacitive sensing buttons A six-segment self- or mutual-capacitive sensing linear slider
PSoC™ 4100S Max pioneer kit (CY8CKIT041S-Max)	Two self- or mutual-capacitive sensing buttons An eight-segment self- or mutual-capacitive sensing linear slider A 10x16 self- or mutual-capacitive sensing touchpad A proximity sensor loop
PSoC™ 4 pioneer kit (CY8CKIT042)	A five-segment linear slider
PSoC™ 4 Bluetooth® LE pioneer Kit (CY8CKIT-042-BLE)	A five-segment linear slider and a wire proximity sensor
PSoC™ 4200-M pioneer kit (CY8CKIT-044)	A five-element gesture detection and two proximity wire sensors
PSoC™ 4200-L pioneer kit (CY8CKIT-046)	A five-element gesture detection, two proximity wire sensors, and an eight-element radial slider
PSoC™ 4100PS prototyping kit (CY8CKIT-147)	No onboard CAPSENSE™ sensors. The kit can be used to connect external sensors to any I/O pin.
CAPSENSE™ proximity shield (CY8CKIT-024)	A four-element gesture detection and one proximity loop sensor
CAPSENSE™ liquid level sensing shield (CY8CKIT-022)	A two-element flexible PCB and 12-element flexible PCB
PSoC™ 4 processor module (CY8CKIT038), with PSoC™ development kit (CY8CKIT001)	A five-segment linear slider and two buttons
CAPSENSE™ expansion board kit (CY8CKIT-031), to be used with CY8CKIT038 and CY8CKIT-001	A 10-segment slider, five buttons and a 4 x 4 matrix button with LED indication.
MiniProg3 program and debug kit (CY8CKIT-002)	CAPSENSE™ performance tuning in CY8CKIT-038
PSoC™ 6 Wi-Fi BT pioneer kit (CY8CKIT-062-WiFi-BT pioneer kit) and PSoC™ 6 Bluetooth® LE pioneer kit (CY8CKIT062-BLE pioneer kit)	A 5-segment CAPSENSE™ slider, two CAPSENSE™ buttons, one CAPSENSE™ proximity sensing header, a proximity sensor.
PSoC™ 6 Wi-Fi BT prototyping kit (CY8CPROTO-063-4343W)	A 5-segment CAPSENSE™ slider and two mutual-cap CAPSENSE™ buttons

~~DETAILED~~ 5 PSoC™ 6 application notes

5.8.5 CAPSENSE™ performance tuning

After you have completed the sensor layout (see [PCB layout guidelines](#)), the next step is to implement the firmware and tune the CAPSENSE™ parameters for the sensor to achieve optimum performance. The CAPSENSE™ sensing method is a combination of hardware and firmware techniques. Therefore, it has several hardware and firmware parameters required for proper operation. These parameters should be tuned to optimum values for reliable touch detection and fast response. Most of the capacitive touch solutions in the market must be manually tuned. A unique feature called SmartSense (also known as Auto-tuning) is available for PSoC™ 4 and PSoC™ 6 CAPSENSE™. SmartSense is a firmware algorithm that automatically sets all parameters to optimum values.

5.8.5.1 Selecting between SmartSense and manual tuning

SmartSense auto-tuning reduces design cycle time and provides stable performance across PCB variations, but requires additional RAM and CPU resources, as indicated in the [Component datasheet/middleware document](#) or [ModusToolbox™ CAPSENSE™ configurator guide](#), to allow runtime tuning of CAPSENSE™ parameters.

SmartSense is recommended mainly for conventional CAPSENSE™ applications involving simple button and slider widgets, and is currently supported only for [Self-capacitance sensing](#) and not [Mutual-capacitance sensing](#).

On the other hand, manual tuning requires effort to tune optimum CAPSENSE™ parameters, but allows strict control over characteristics of capacitive sensing system, such as response time and power consumption. It also allows use of CAPSENSE™ beyond the conventional button and slider applications such as proximity and liquid-level-sensing.

SmartSense is the recommended tuning method for all the conventional CAPSENSE™ applications. You should use SmartSense auto-tuning if your design meets the following requirements:

- The design is for conventional user-interface application like buttons, sliders, and touchpad
- The parasitic capacitance (C_p) of the sensors is within SmartSense-supported range as mentioned in the “SmartSense operating conditions” section in [Component datasheet/middleware document](#) or [ModusToolbox™ CAPSENSE™ configurator guide](#)
- The sensor scan time chosen by SmartSense meets the response time/power requirements of the end system
- SmartSense auto-tuning meets the RAM/flash requirements of the design

For all other applications, use [Manual tuning](#). In such cases, you can also use SmartSense as an initial step to find the optimum hardware parameters such as Sense Clock frequency, and then change the tuning mode to manual tuning for further tuning of the CapSense parameters. See [Using SmartSense to determine hardware parameters](#).

Note: *Manual tuning requires I²C or UART communication with a host PC.*

5.8.5.2 SmartSense

5.8.5.2.1 Overview

The CAPSENSE™ algorithm is a combination of hardware and firmware blocks inside PSoC™. Therefore, it has several hardware and firmware parameters required for proper operation. These parameters need to be tuned to optimum values for reliable touch detection and fast response.

SmartSense is a CAPSENSE™ tuning method that automatically sets sensing parameters for optimal performance, based on user-specified finger capacitance values, and continuously compensates for system, manufacturing, and environmental changes.

5 PSoC™ 6 application notes

DRAFT

Note: SmartSense currently supports widgets with **CSD (Self-cap)** Sensing mode only. **CSX (Mutual-cap)** widgets must be tuned manually.

Some advantages of SmartSense, as opposed to manual tuning are:

- **Reduced design cycle time:** The design flow for capacitive touch applications involves tuning all of the sensors. This step can be time consuming if there are many sensors in your design. In addition, you must repeat the tuning when there is a change in the design, PCB layout, or mechanical design. Auto-tuning solves these problems by setting all of the parameters automatically. [Figure 244](#) shows the design flow for a typical CAPSENSE™ application with and without SmartSense

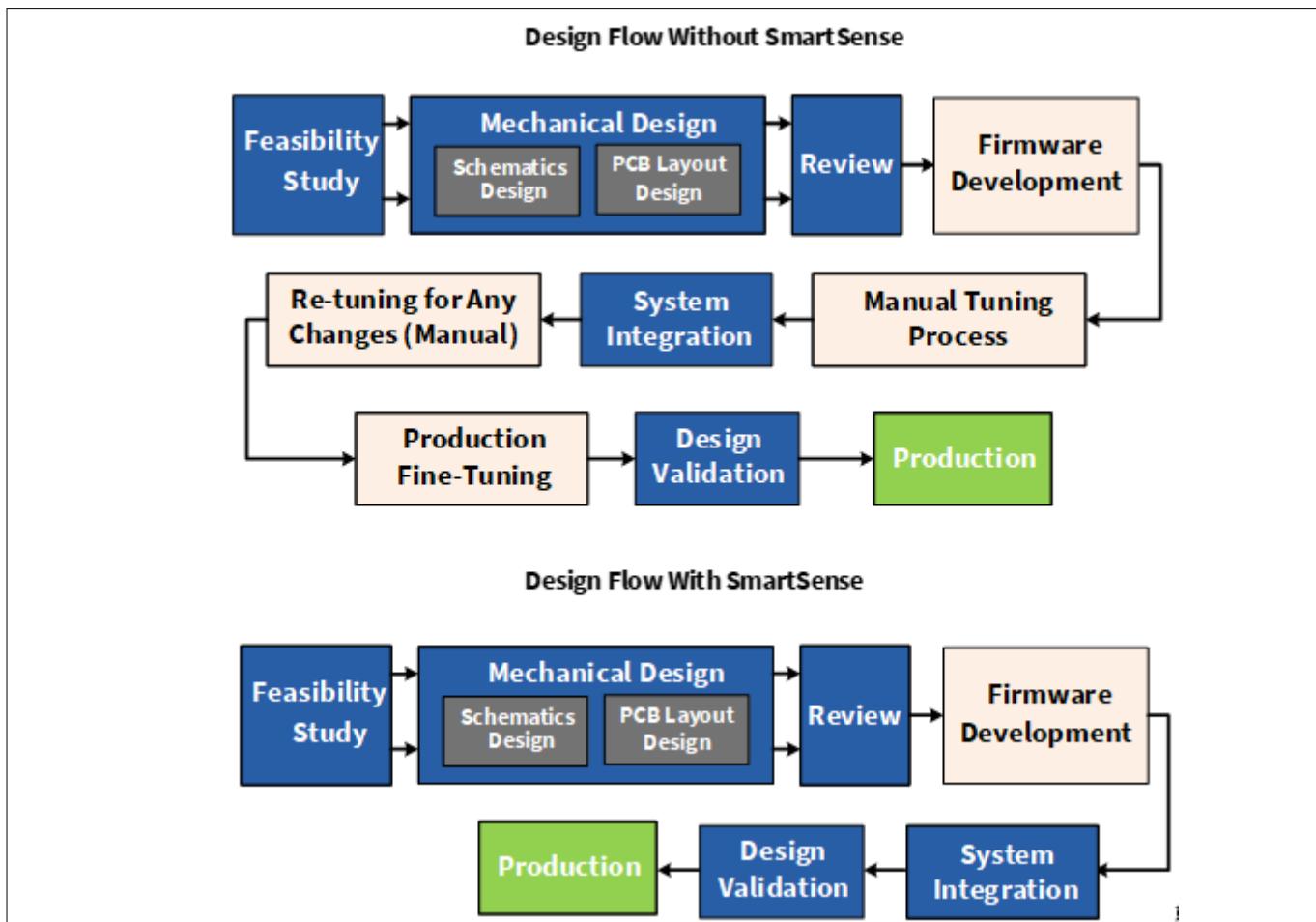


Figure 244 Design flow with and without SmartSense

- **Performance is independent of PCB variations:** The parasitic capacitance, C_p , of individual sensors can vary due to process variations in PCB manufacturing, or vendor-to-vendor variation in a multi-sourced supply chain. If there is significant variation in C_p across product batches, the CAPSENSE™ parameters must be re-tuned for each batch. SmartSense sets parameters for each device automatically, hence taking care of variations in C_p
- **Ease of use:** SmartSense is faster and easier to use because only a basic knowledge of CAPSENSE™ is needed

Note that SmartSense can be used in multiple ways:

1. SmartSense (Full auto-tune) – This is the quickest way to tune. This method calibrates CAPSENSE™ hardware and software parameters automatically at runtime. This is the recommended method for most designs
2. SmartSense (Hardware parameters only) – This method auto-tunes all hardware parameters of CAPSENSE™, but allows to set user-defined threshold values (see [Table 45](#)). This method consumes less

~~5 PSoC™ 6 application notes~~

flash/RAM resources than SmartSense (Full Auto-Tune). Also, this method avoids the extra processing needed for automatic threshold calculation and hence allows lower power consumption for a given scan rate. Use this method for low-power or noisy designs or in cases with constrained memory requirements.

- 3.** SmartSense for initial tuning – You may also use SmartSense for initial tuning, to quickly find the best settings for a CAPSENSE™ board and then change to manual tuning. This method is useful for cases with strict requirements on response time or power consumption. This is a quick method to find the best settings, instead of starting manual tuning from scratch. Refer to the section [Using SmartSense to determine hardware parameters](#) for more details

4. **Table 40 CAPSENSE™ parameters auto-tuned in SmartSense**

Parameter	Full auto-tune mode	Hardware parameters only mode
Scan resolution	Calculated once on CAPSENSE™ initialization.	
Compensation IDAC		
Modulator IDAC		
Sense clock frequency		
Modulator clock frequency		
Finger threshold	Calculated once on CAPSENSE™ initialization based on the selected finger capacitance and updated after each sensor scan.	Manual selection (see Table 45).
Noise threshold		
Hysteresis		
Negative noise threshold		
Low baseline reset		

5.8.5.2.2 SmartSense full auto-tune

In SmartSense Full Auto-tune mode, the only parameter that needs to be tuned by the user is the Finger Capacitance parameter. The Finger Capacitance parameter (C_F) indicates the minimum value of finger capacitance that should be detected as a valid touch by the CAPSENSE™ Component. Whenever the actual C_F that is added when the finger touches the button sensor is greater than the value specified for the Finger Capacitance parameter in the Component configuration window, the sensor status will change to '1'; however, if the actual C_F added by the finger touch is less than the value specified in the Component configuration window, the sensor status will remain '0'. The way of tuning the finger capacitance is different for button and slider widgets.

Note: Even for SmartSense auto-tuning, the CAPSENSE™ Component allows manual configuration of some general parameters like enable/disable of compensation IDAC, filters, shield such as liquid-tolerance-related parameters and modulator clock. These can be left at their default values for most cases or configured based on the respective sections in this guide.

Tuning button widgets

This section explains how to choose the Finger capacitance value for the Button widget. You may perform only a coarse tuning of the Finger capacitance parameter for a working design, or you may choose to fine-tune the Finger capacitance value. Coarse-tuning will satisfy the requirements of most designs, but fine-tuning will allow you to choose the most efficient CAPSENSE™ parameters (that is, minimum sensor scan time) using SmartSense.

If you do not know the value of C_F (C_F can be estimated based on [Equation 2](#)), set the Finger capacitance as follows:

5 PSoC™ 6 application notes

- ~~1.~~ Start by specifying the highest value for finger capacitance (from the available options in the list) and check the SNR and button status when the button is touched. Use the [Tuner GUI](#) to find the SNR
- ~~2.~~ Decrease the finger capacitance parameter value until the button status changes to '1' on touch and $\text{SNR} > 5$. [Figure 245](#) shows the detailed steps to find the right value for the Finger capacitance parameter in your design

Enable filters if the SNR of one or more sensors is less than 5:1 when the set finger capacitance is already at the least finger capacitance supported in the Component. You can also enable filters if externally induced noise is causing a decrease in SNR. See [Table 41](#) to choose the right filter in this case. There are various types of filters available in the CAPSENSE™ Component such as Median Filter, IIR filter, and Average Filter; you can enable more than one filter to reduce the noise in the raw count according to the requirement.

If you choose to use an IIR filter, begin by selecting a filter with a higher value of the filter coefficient and keep decreasing it until you achieve an SNR greater than or equal to 5:1. Using filters will affect the response time. You must properly select the filter coefficient such that the response time and SNR requirement are satisfied.

If the SNR is still less than 5:1 even when the smallest allowed value of finger capacitance and proper filter is chosen, see [PCB layout](#), [Manual tuning](#), or [Tuning debug FAQs](#) for more details on debugging the issue.

5 PSoC™ 6 application notes

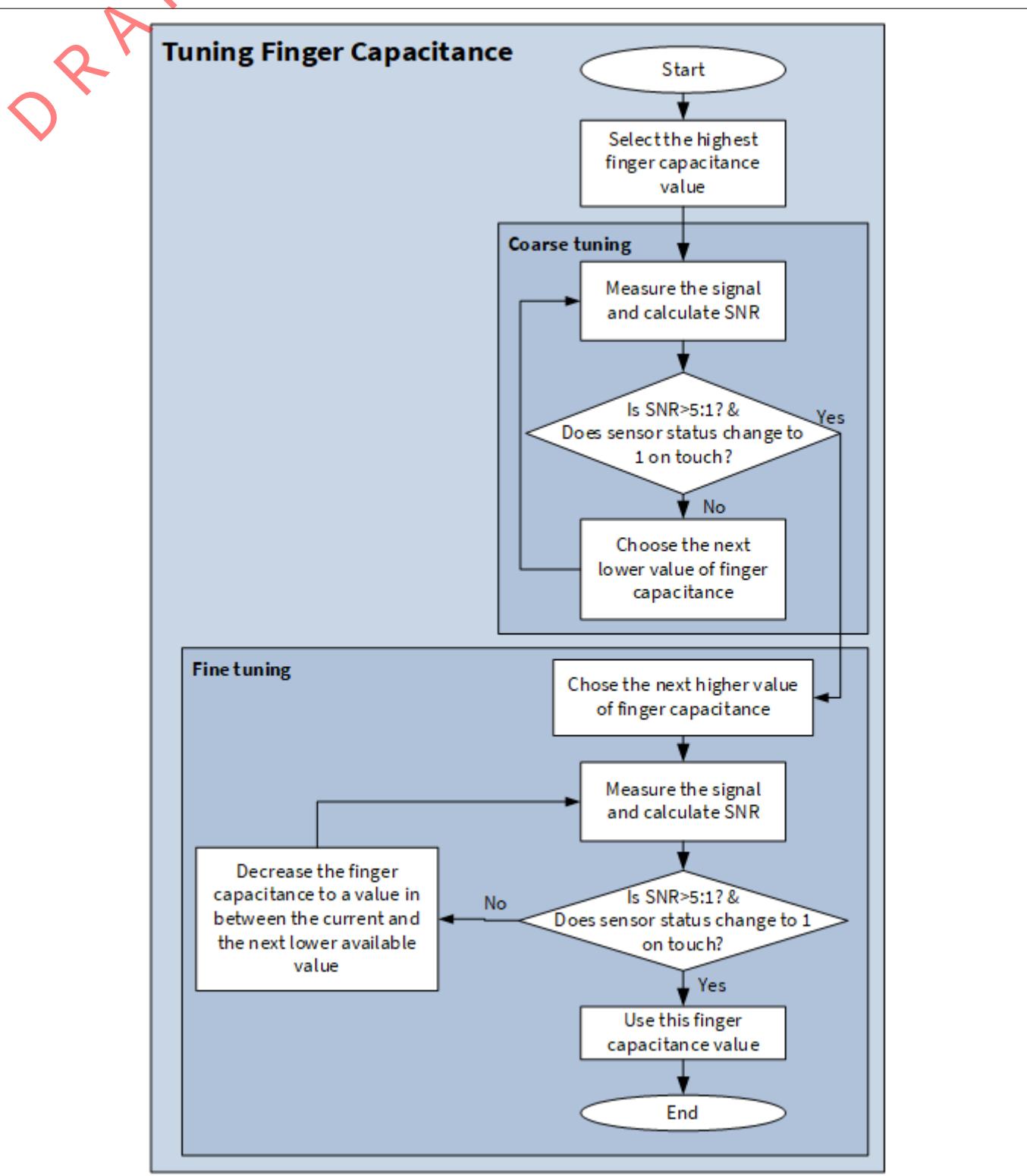


Figure 245 Using SmartSense auto-tuning based CAPSENSE™ project in PSoC™ Creator

5 PSoC™ 6 application notes

~~DRAFT~~ Table 41 Raw data noise filters in CAPSENSE™ component

Filter	Description	Mathematical description	Application
Median	Nonlinear filter that takes the three most recent samples and computes the median value.	$y[i] = \text{median}(x[i], x[i - 1], x[i - 2])$	Eliminates noise spikes from motors and switching power supplies
Average	Finite impulse response filter (no feedback) with equally weighted coefficients. It takes the four most recent samples and computes their average.	$y[i] = \frac{1}{4} \times (x[i] + x[i - 1] + x[i - 2] + x[i - 3])$	Eliminates periodic noise (for example, from power supplies)
First Order IIR	Infinite impulse response filter (feedback) with a step response similar to an RC low pass filter, thereby passing the low-frequency signals (finger touch responses). K value is fixed to 256. N is the IIR filter raw count coefficient. A lower N value results in lower noise, but slows down the response.	$y[i] = \frac{1}{K} \times \{N \times x[i] + (K - N) \times y[i - 1]\}$	Eliminates high frequency noise.

Tuning slider widgets

For sliders, set finger capacitance to the highest value initially. Slide your finger on the slider. If at any position on the slider, at least one slider segment status is ON and has an SNR >5:1, and at least two slider segments report a “difference count” that is, a “sensor signal” value greater than 0, use this finger capacitance value. Otherwise, decrease the finger capacitance value until the above condition holds true. [Figure 246](#) shows how to tune the finger capacitance for slider widget.

If these conditions are not met even after setting minimum allowed Finger Capacitance, use [Manual tuning](#) or revise the hardware according to [Slider design](#) considerations or see [Tuning debug FAQs](#). [Figure 246](#) explains the process of setting finger capacitance value for sliders.

Note: *It is recommended to use the compensation IDAC because it allows a higher variation in the parasitic capacitance of the slider segment with respect to the slider segment that has the maximum C_P.*

5 PSoC™ 6 application notes

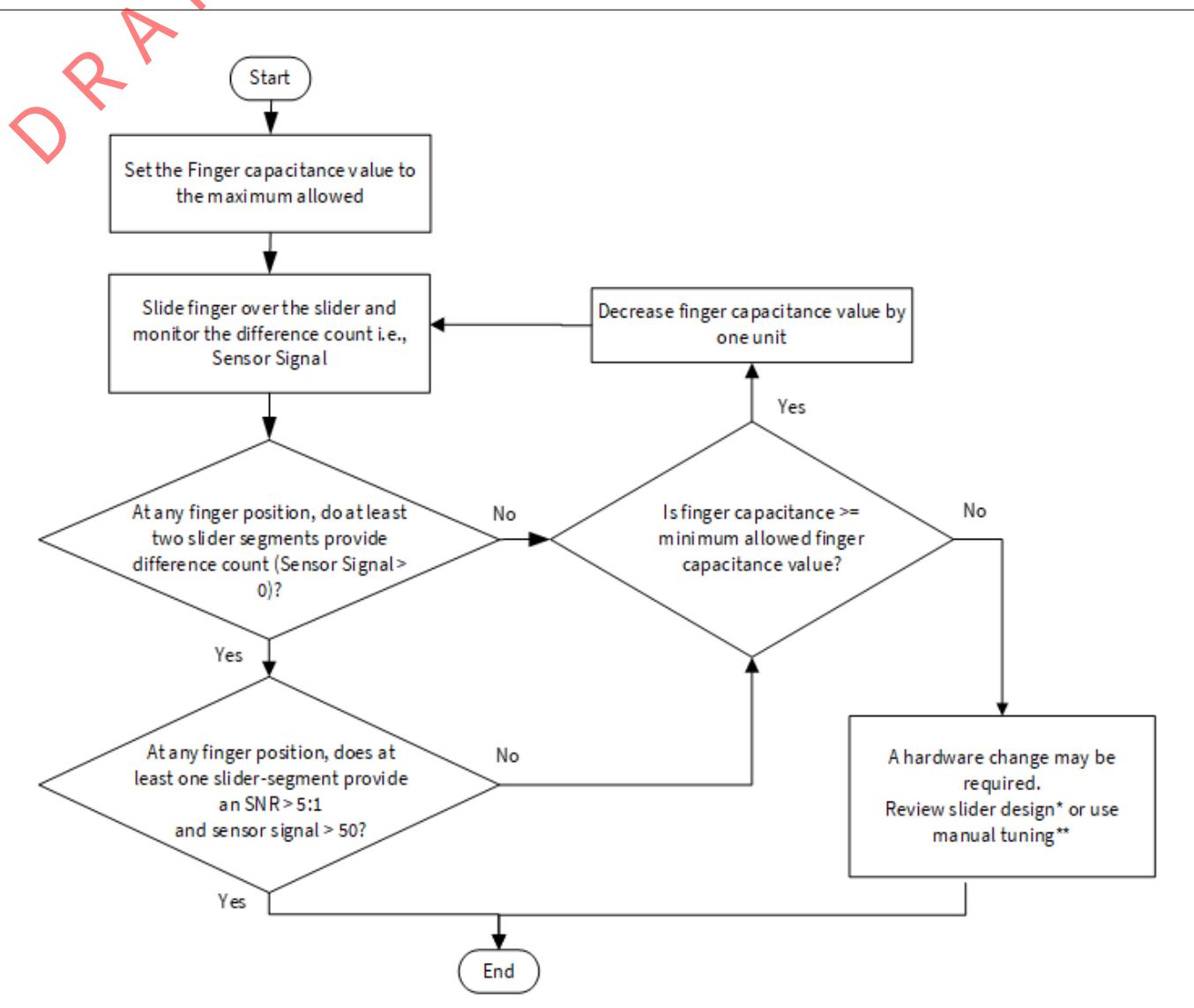


Figure 246 Setting finger capacitance value for ssiders

* To review slider design, see the [Slider design](#) section in the [Design considerations](#) chapter.

** To do manual tuning, see the [Manual tuning](#) section in the [CAPSENSE™ performance tuning](#) chapter.

Tuning proximity widgets

See [AN92239 Proximity sensing with CAPSENSE™](#) and the “Proximity sensing” section in [Getting started with CAPSENSE™ design guide](#).

5.8.5.2.3 SmartSense hardware parameters-only mode

See [Table 45](#) for the recommended values for thresholds when the CSD tuning method is SmartSense (Hardware parameters only).

5.8.5.2.4 SmartSense for initial tuning

See [Using SmartSense to determine hardware parameters](#) for more details.

5 PSoC™ 6 application notes**5.8.5.3 Manual tuning****5.8.5.3.1 Overview**

SmartSense technology allows a device to calibrate itself for optimal performance and complete the entire tuning process automatically. This technology will meet the needs of most designs, but in cases where SmartSense does not work or there are specific SNR or power requirements, the CAPSENSE™ parameters can be adjusted to meet system requirements. This can be achieved by manual tuning.

Some advantages of manual tuning, as opposed to [SmartSense auto-tuning](#) are:

- Strict control over parameter settings: SmartSense sets all the parameters automatically. However, there may be situations where you need to have strict control over the parameters. For example, use manual tuning if you need to strictly control the time PSoC™ takes to scan a group of sensors or strictly control the sense clock frequency of each sensor (this can be done to reduce EMI in systems)
- Supports higher parasitic capacitances: If the parasitic capacitance is higher than the value supported by SmartSense, you should use manual tuning. See the [Component datasheet/middleware document](#) for more details on the supported range of parasitic capacitance by SmartSense

The manual tuning process can be summarized in the following three steps and is shown in [Figure 247](#).

Set initial values of [Selecting CAPSENSE™ hardware parameters](#) using SmartSense (see [Using SmartSense to determine hardware parameters](#)) or determine the values manually.

Tune CAPSENSE™ component hardware parameters to ensure that [Signal-to-noise](#) is greater than 5:1 with a signal of at least 50 counts while meeting the system timing requirements.

Set optimum values of [Selecting CAPSENSE™ software parameters](#).

The following sections describe the fundamentals of manual tuning and the above three steps in detail. Knowledge of the CAPSENSE™ architecture in PSoC™ is a prerequisite for these sections. See [Capacitive touch sensing method](#) and [CAPSENSE™ generations in PSoC™ 4 and PSoC™ 6](#). The main difference in CAPSENSE™ architecture across different generations are listed in [Table 35](#).

Depending upon the sensing method selected, the manual tuning procedure will differ. See [CSD sensing method \(third- and fourth-generation\)](#), [CSX sensing method \(third- and fourth-generation\)](#) chapter for their respective manual tuning procedures. You can skip these sections if you are not planning to use manual tuning in your design. [Figure 247](#) shows a general manual tuning procedure.

5 PSoC™ 6 application notes

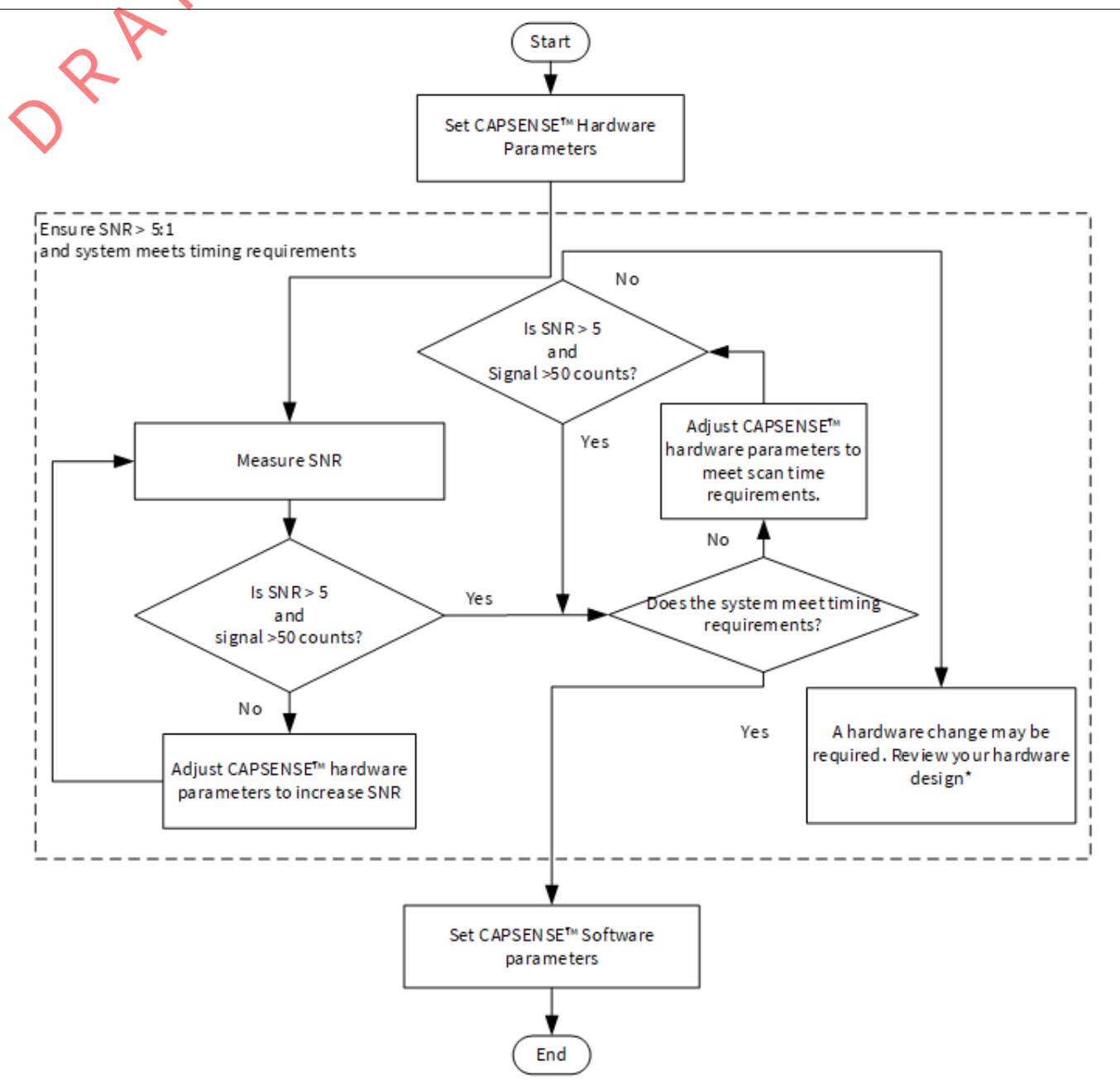


Figure 247 Manual tuning process overview

* To review the hardware design, see the [Sensor construction](#) and [PCB layout guidelines](#) sections in the [Design considerations](#) chapter.

5.8.5.3.2 CSD sensing method (third- and fourth-generation)

This section explains the basics of manual tuning using CSD sensing method. It also explains the hardware and software parameters that influence CSD sensing method and procedure of manual tuning for button, slider, touchpad and proximity widgets.

5 PSoC™ 6 application notes

Basics

Conversion gain and CAPSENSE™ signal

Conversion gain will influence how much signal the system sees for a finger touch on the sensor. If there is more gain, the signal is higher, and a higher signal means a higher achievable **Signal-to-noise ratio (SNR)**.

Note: *An increased gain may result in an increase in both signal and noise. However, if required, you can use firmware filters to decrease noise. For details on available firmware filters, see [Table 41](#).*

Conversion gain in single IDAC mode

In the [single IDAC mode](#), the raw count is directly proportional to the sensor capacitance.

$$\text{rawcount} = G_{CSD} C_S$$

Equation 25 Raw count relationship to sensor capacitance

Where,

C_S = sensor capacitance

$C_S = C_P$ if there is no finger present on sensor

$C_S = (C_P + C_F)$ when there is a finger present on the sensor

G_{CSD} = Capacitance to digital conversion gain of CAPSENSE™ CSD

The approximate value of this conversion gain using the IDAC sourcing mode, according to [Equation 11](#) and [Equation 25](#) is:

$$G_{CSD} = (2^N - 1) \cdot \frac{V_{REF} F_{SW}}{I_{MOD}}$$

Equation 26 Capacitance to digital converter gain

$$G_{CSD} = (2^N - 1) \cdot \frac{(V_{DD} - V_{REF}) F_{SW}}{I_{MOD}}$$

Equation 27 Capacitance to digital converter gain (sinking IDAC mode)

Where,

V_{REF} = Comparator reference voltage. Refer [Table 35](#).

F_{SW} = Sense clock frequency

I_{MOD} = Modulator IDAC current

N = Resolution of the sigma to delta converter.

The tunable parameters of the conversion gain are V_{REF} , F_{SW} , I_{MOD} , and N. [Figure 248](#) illustrates a plot of raw count versus sensor capacitance.

5 PSoC™ 6 application notes

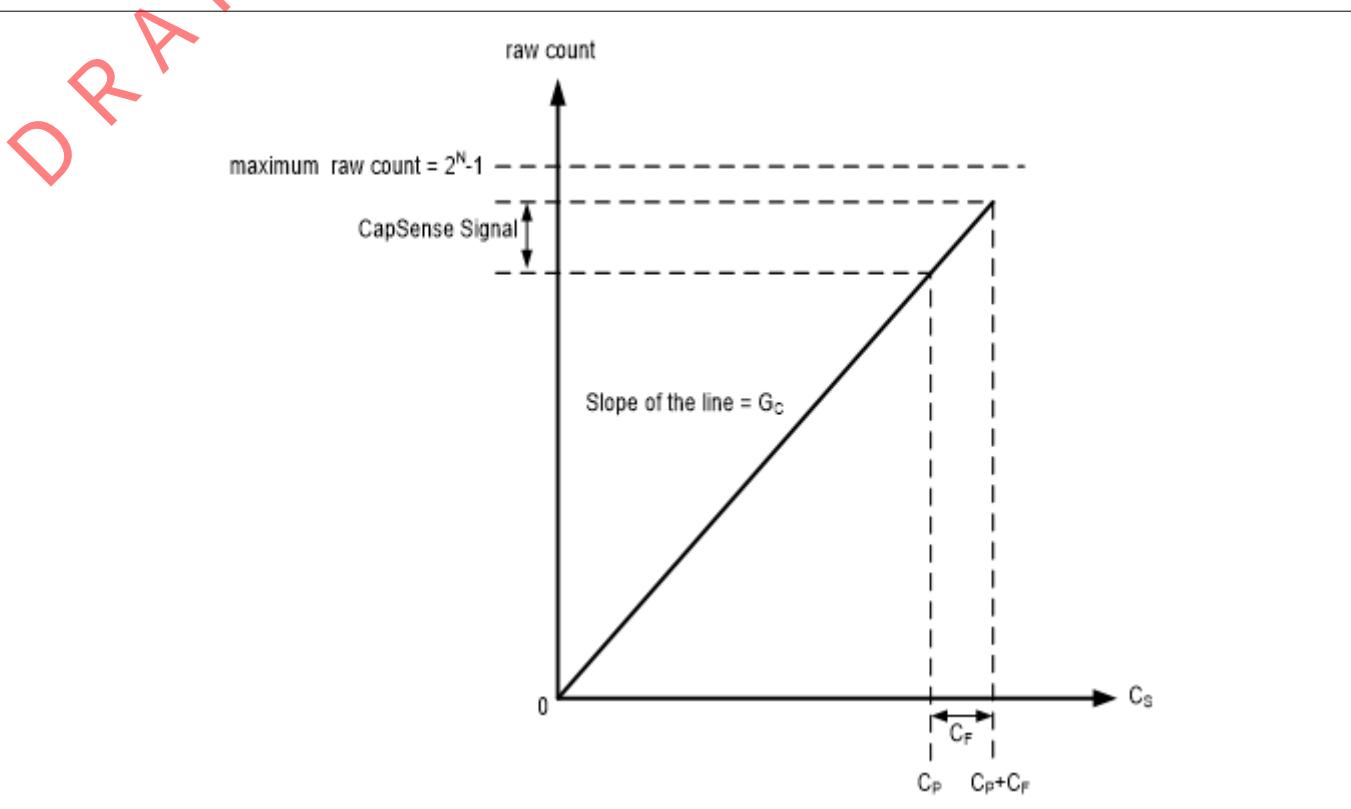


Figure 248 Raw count versus sensor capacitance

The change in raw counts when a finger is placed on the sensor is called CAPSENSE™ signal. Figure 249 shows how the value of the signal changes with respect to the conversion gain.

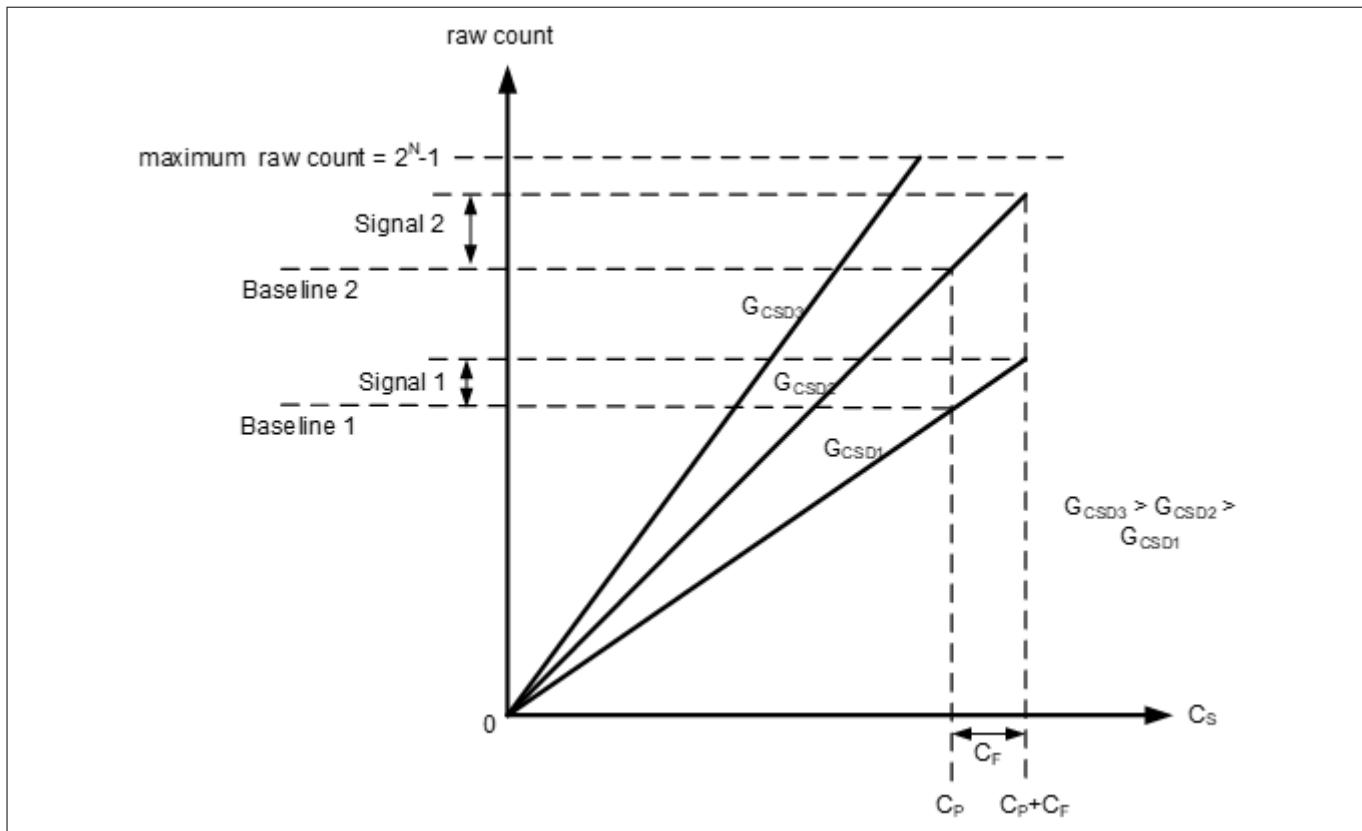


Figure 249 Signal values for different conversion gains

5 PSoC™ 6 application notes

~~DRAFT~~

Figure 249 shows three plots corresponding to three conversion gain values G_{CSD3} , G_{CSD2} , and G_{CSD1} . An increase in the conversion gain results in higher signal value. However, this increase in the conversion gain also moves the raw count corresponding to C_P (that is, Baseline) towards the maximum value of raw count (2^N-1). For very high gain values, the raw count saturates as the plot of G_{CSD3} shows. Therefore, you should tune the conversion gain to get a good signal value while avoiding saturation of raw count. Tune the CSD parameters such that when there is no finger on the sensor, that is when $C_S = C_P$, the raw count = 85% of (2^N-1) as Figure 250 shows. This ensures maximum gain, with enough margin for the raw count to grow because of environmental changes, and not saturate on finger touches.

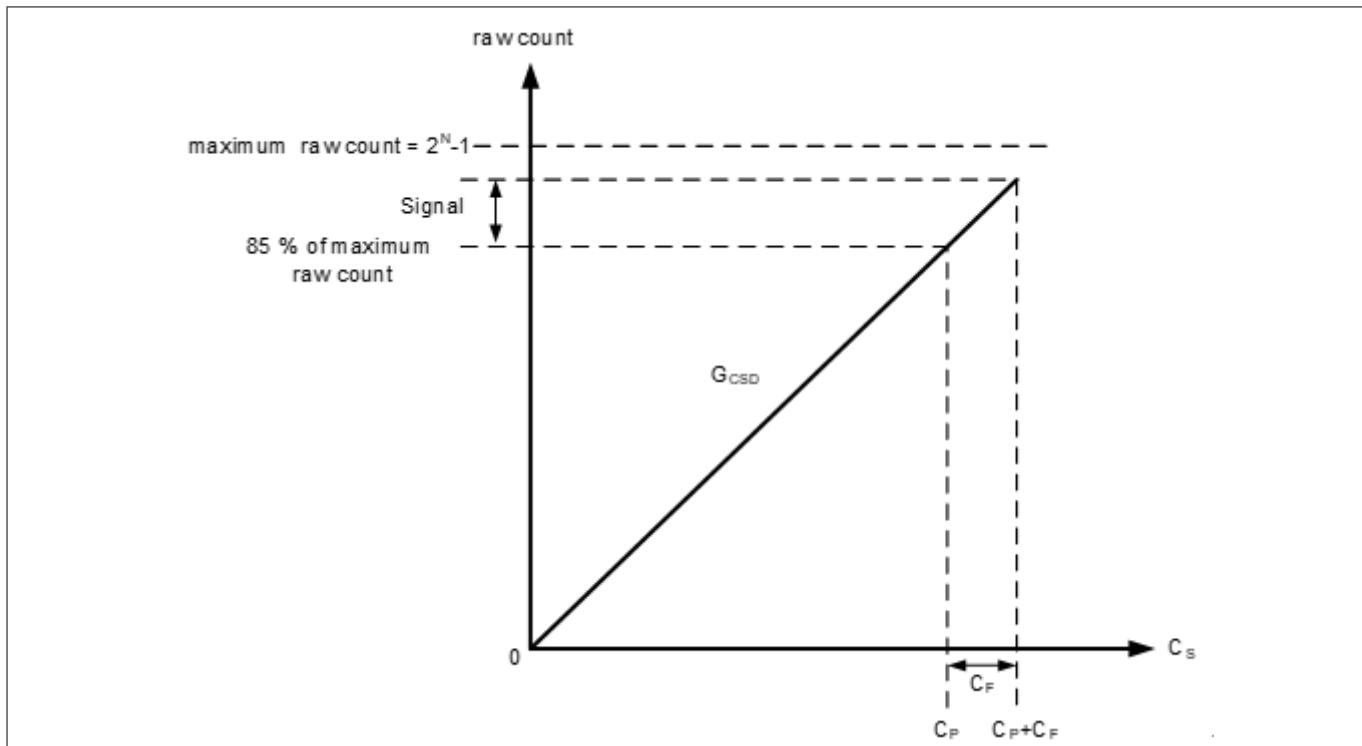


Figure 250 Recommended tuning

Conversion gain in dual IDAC mode

The equation for raw count in the [dual IDAC mode](#), according to [Equation 25](#) and [Equation 13](#) is:

$$\text{rawcount} = G_{CSD}C_S - \left(2^N - 1\right) \frac{I_{COMP}}{I_{MOD}}$$

Equation 28 Dual IDAC mode raw counts

Where,

I_{COMP} = Compensation IDAC current

G_{CSD} is given by [Equation 10](#) for sourcing IDAC mode and [Equation 27](#) for sinking IDAC mode.

In both single IDAC and dual IDAC mode, tune the CSD parameters, so that when there is no finger on the sensor, that is when $C_S = C_P$ the raw count = 85% of (2^N-1), as [Figure 251](#) shows, to ensure high conversion gain, to avoid [Flat-spots](#), and to avoid raw count saturation due to environmental changes.

DRAFT

5 PSoC™ 6 application notes

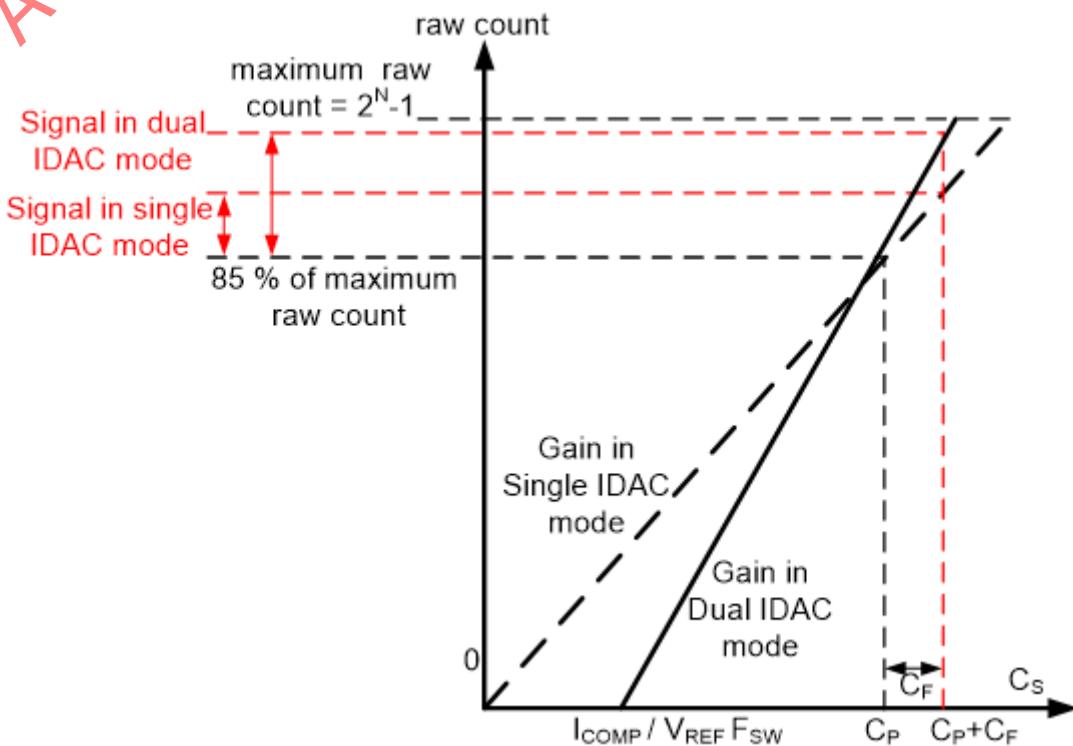


Figure 251 Recommended tuning in dual IDAC mode

As Figure 251 shows, the 85% requirement restricts to a fixed gain in single-IDAC mode, while in dual-IDAC mode, gain can be increased by moving the C_S axis intercept to the right (by increasing I_{COMP}) and correspondingly decreasing the modulator IDAC (I_{MOD}) to still achieve raw count = 85% of (2^N-1) for $C_S = C_P$. Using dual IDAC mode this way brings the following changes to the Raw Count versus C_P graph:

1. Use of compensation IDAC introduces a non-zero intercept on the C_S axis as given in .

$$C_S \text{ axis intercept} = \left(\frac{I_{COMP}}{V_{REF} F_{sw}} \right)$$

Equation 29 C_S axis intercept with regards to I_{COMP}

The value of I_{MOD} in the dual IDAC mode is half compared to the value of I_{MOD} in the single IDAC mode (all other parameters remaining the same), so the gain G_{CSID} in the dual IDAC mode is double the gain in the single IDAC mode according to [dual IDAC mode](#). Thus, the signal in the dual IDAC mode is double the signal in the single IDAC mode for a given resolution N.

While manually tuning a sensor, keep [dual IDAC mode](#) and [Equation 23](#) as well as the following points in mind:

1. Higher gain leads to increased sensitivity and better overall system performance. However, do not set the gain such that raw counts saturate, as the plot of gain G_{CSID} shows in [Figure 249](#). It is recommended to set the gain in such a way that the raw count corresponding to C_P is 85 percent of the maximum raw count for both the single IDAC and dual IDAC mode. The sense clock frequency (F_{sw}) should be set carefully; higher the frequency, higher the gain, but the frequency needs to be low enough to fully charge and discharge the sensor as [Equation 23](#) indicates
2. Enabling the Compensation IDAC plays a huge role in increasing the gain; it will double the gain if set as recommended [above](#). Always enable the Compensation IDAC when it is not being used for general-purpose applications

~~5 PSoC™ 6 application notes~~

- ~~DRAFT~~
3. Lower the modulation IDAC current, higher the gain. Adjust your IDAC to achieve the highest gain, but make sure that the raw counts corresponding to C_p have enough margin for environmental changes such as temperature shifts, as indicated in [Figure 250](#) and [Figure 251](#)
 4. Increasing the number of bits of resolution used for scanning increases gain. An increase in resolution by one bit will double the gain of the system, but also double the scan time according to [Equation 9](#). A balance of scan time and gain needs to be achieved using resolution

Flat-spots

Ideally, raw counts should have a linear relationship with sensor capacitance as [Figure 248](#) and [Figure 251](#) show. However, in practice, sigma delta modulators have non-sensitivity zones, also called flat-spots or dead-zones – for a range of sensor capacitance values, the sigma delta modulator may produce the same raw count value as [Figure 252](#) shows. This range is known as a dead-zone or a flat-spot.

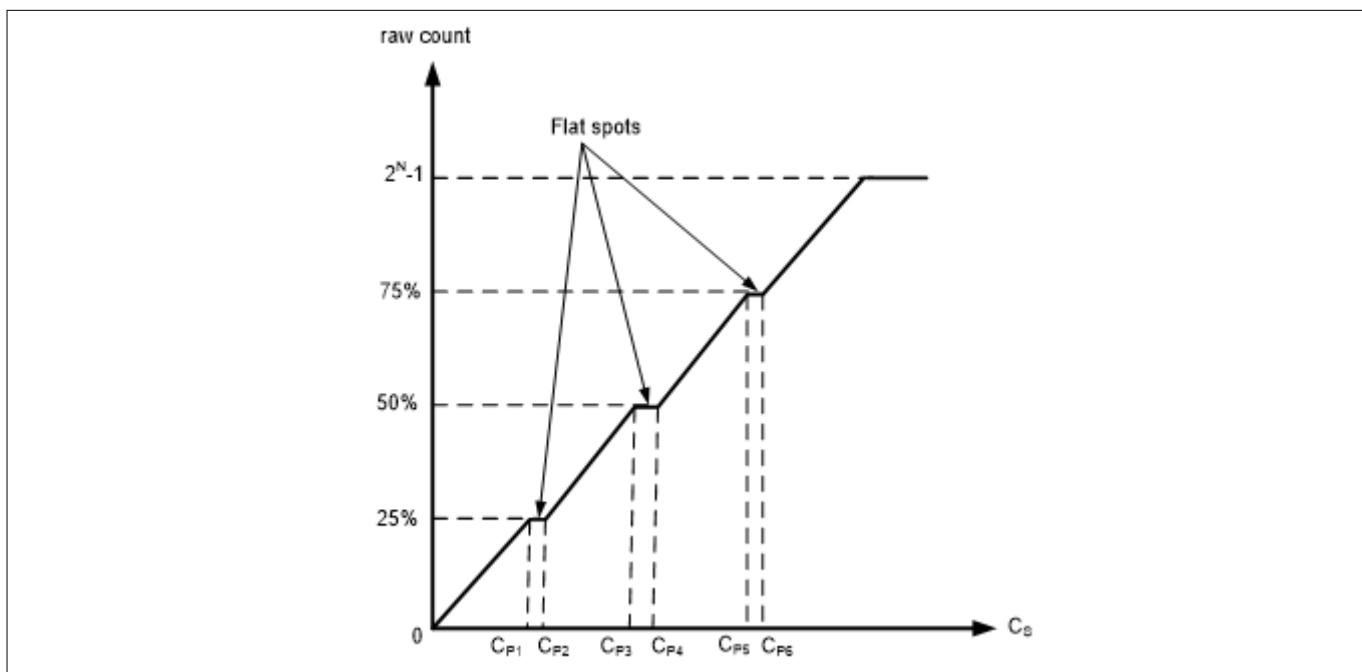


Figure 252 Flat-spots in raw counts versus sensor capacitance when direct clock is used

In the case of CAPSENSE™ CSD, these flat spots occur near 25, 50, and 75 percent of the maximum raw count value (that is, near 25%, 50%, and 75% of 2^N-1 , where $N = \text{Scan resolution}$). These flat spots are prominent when direct clock is used as [Sense Clock](#) source. Flat-spots do not occur if PRS is used as the Sense Clock source (see also section [Using SmartSense to determine hardware parameters](#)).

For almost all systems, we recommend using PRS as the Sense Clock source because it limits the impact of flat spots and also provides EMI/EMC benefits as indicated in [Sense Clock](#). If your system requires a direct clock, ensure that you use [auto-calibration](#) or avoid this raw count range when using manual calibration.

Flat-Spots Reduction Techniques

1. Calibrate rawcount to 85%

In the case of CAPSENSE™ CSD, these flat-spots occur near 25, 50, and 75 percent of the maximum raw count value (that is, near 25%, 50%, and 75% of 2^N-1 , where ' N ' is the Scan resolution). Setting calibration to 85% decrease the width of flat-spots significantly.

2. Use PRS clock

These flat-spots are prominent when direct clock is used as [Sense Clock](#) source. Flat-spots do not occur if PRS is used as the Sense Clock source (see also section [Using SmartSense to determine hardware](#)

~~5 PSoC™ 6 application notes~~

~~DRAFT~~ parameters. For almost all systems, we recommend using PRS as the Sense Clock source because it limits the impact of flat-spots and also provides EMI/EMC benefits as indicated in [Sense clock source](#) Sense clock . If your system requires a direct clock, ensure that you use [auto-calibration](#) or avoid this raw count range when using manual calibration.

Selecting CAPSENSE™ hardware parameters

CAPSENSE™ hardware parameters govern the conversion gain and CAPSENSE™ signal. [Table 42](#) lists the CAPSENSE™ hardware parameters that apply to CSD sensing method. The following subsection gives guidance on how to adjust these parameters to achieve optimal performance for CAPSENSE™ CSD system.

For simplicity of documentation, this design guide shows selecting the CAPSENSE™ parameters in PSoC™ Creator. You can use the same procedure to set the parameters in ModusToolbox™. However, in ModusToolbox™, you set the Sense clock and Modulator clock using divider values while in the PSoC™ Creator you specify the frequency value directly in the configurator. For more details on configuring CAPSENSE™, see the [Component datasheet/middleware](#) document.

Table 42 CAPSENSE™ component hardware parameters

Sl. No.	CAPSENSE™ parameter in PSoC™ Creator	CAPSENSE™ parameter in ModusToolbox™
1	Sense clock frequency	Sense clock divider
2	Sense clock source	Sense clock source
3	Modulator clock frequency	Modulator clock divider
4	Modulator IDAC	Modulator IDAC
5	Compensation IDAC	Compensation IDAC
6	Scan resolution	Scan resolution

Using SmartSense to determine hardware parameters

Parameters listed in [Table 42](#) are CAPSENSE™ hardware parameters. Tuning these parameters manually for optimal value is a time-consuming task. You can use SmartSense to determine these hardware parameters and take it as an initial value for manual tuning. You can fine-tune these values to further optimize the scan time, SNR, power consumption, or improving EMI/EMC capability of the CAPSENSE™ system.

Set the tuning mode to SmartSense and configure default values for parameters other than finger capacitance. See the [SmartSense](#) section for the tuning procedure and use the Tuner GUI to read back all the hardware parameters set by SmartSense. See the [Component datasheet/middleware document](#) for more details on how to use the Tuner GUI.

[Figure 253](#) shows the best hardware parameter values in the Tuner GUI that are tuned by SmartSense for a specific hardware to sense a minimum finger capacitance of 0.1 pF.

5 PSoC™ 6 application notes

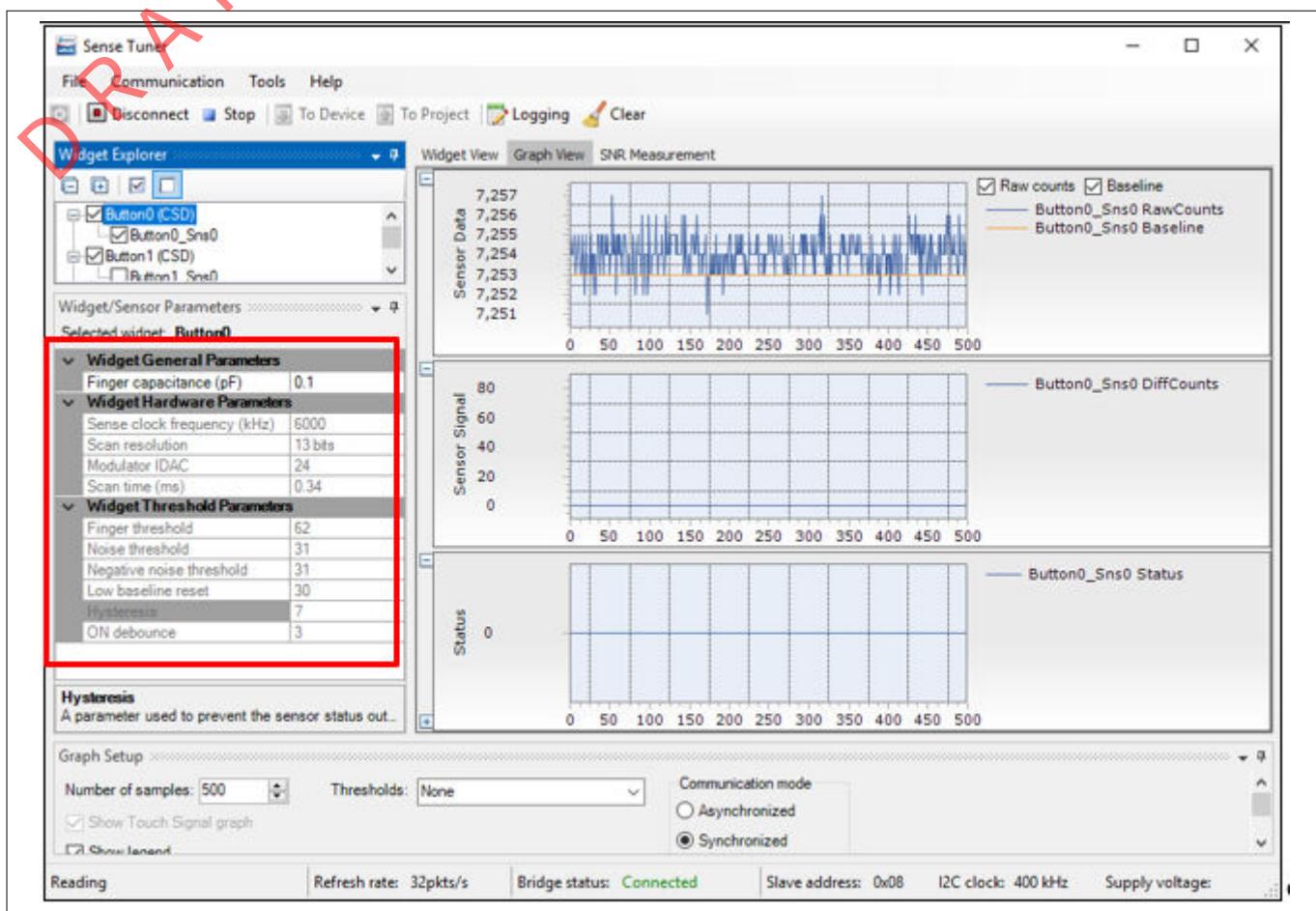


Figure 253 Read-back hardware parameter values in Tuner GUI

Manually tuning hardware parameters

Sense clock parameters

There are two parameters that are related to Sense clock: Sense clock source and Sense clock frequency.

Sense clock source

Select “Auto” to let the Component automatically choose the best Sense clock source from Direct, PRSx, and SSCx for each widget. If not selecting Auto, select the clock source based on the following:

- Use pseudo random sequence (PRSx) modes to remove flat-spots
- Use spread spectrum clock (SSCx) modes for reducing EMI/EMC noise at a particular frequency. This feature is available in PSoC™ 4 S-Series, PSoC™ 4100S Plus, PSoC™ 4100PS, and PSoC™ 6 family of devices. In this case, the frequency of the sense clock is spread over a predetermined range
- Use Direct clock for absolute capacitance measurement

When selecting PRSx as the sense clock source, ensure that the sequence completes within one conversion cycle; not letting the sequence complete may cause high noise in raw count. that is, $T_{PRS} \ll T_{SCAN}$.

For PRS clock, use the following equations to calculate one PRS sequence completion cycle and scan time.

5 PSoC™ 6 application notes

$$T_{SCAN} = \frac{2^N - 1}{F_{MOD}}, \text{ here } N \text{ is the Scan resolution}$$

Equation 30 Sensor scan time

$$T_{PRS} = \frac{2^{N_PRS} - 1}{F_{SW}}, \text{ here } N_PRS \text{ is either 8 or 12}$$

Equation 31 PRS sequence period

See the [Component datasheet/middleware document](#) for more details on the rules and recommendations for SSCx selection.

Sense clock frequency

The sense clock frequency should be selected so that the sensor will charge and discharge completely in each sense clock period as [Figure 229](#) shows.

This requires that the maximum sense clock frequency be chosen per [Equation 31](#).

$$F_{SW}(\text{maximum}) = \frac{1}{10R_{SeriesTotal}C_P}$$

Equation 32 Sense clock maximum frequency

$$R_{SeriesTotal} = R_{EXT} + R_{GPIO}$$

Equation 33 Total series resistance

Here, C_P is the sensor parasitic capacitance, and $R_{SeriesTotal}$ is the total series-resistance, including the $500\ \Omega$ resistance of the internal switches, the recommended external series resistance of $560\ \Omega$ (connected on PCB trace connecting sensor pad to the device pin), and trace resistance if using highly resistive materials (example ITO or conductive ink); that is, a total of $1.06\ k\Omega$ plus the trace resistance.

The value for C_P can be estimated using the CSD Built-in-Self-test API; `GetSensorCapacitance()`. See the [Component datasheet/middleware document](#) for details.

[Equation 26](#) shows that it is best to use the maximum clock frequency to have a good gain; however, you should ensure that the sensor capacitor fully charges and discharges as shown in [Figure 229](#).

Generally, the C_P of the shield electrode will be higher compared to sensor C_P . For good liquid tolerance, the shield signal should satisfy the condition mentioned in [Shield electrode tuning theory](#). If it is not satisfied, reduce the sense clock frequency further to satisfy the condition.

Modulator clock frequency

The modulator clock governs the conversion time for capacitance-to-digital conversion, also called the “sensor scan time” (see [Equation 9](#)).

A lower modulator clock frequency implies the following:

Longer conversion time (see [Equation 23](#) and [Equation 21](#))

- Lower peak-to-peak noise on raw count because of longer integration time of the sigma-delta converter
- Wider [Flat-spots](#)

~~5 PSoC™ 6 application notes~~

Select the highest frequency for the shortest conversion time and narrower flat spots for most cases. Use slower modulator clock to reduce peak-to-peak noise in raw counts if required.

Modulation and compensation IDACs

CSD supports two IDACs: Modulation IDAC and Compensation IDAC that charge C_{MOD} as [Figure 225](#) shows. These govern the [Conversion gain in dual IDAC mode](#) for capacitance-to-digital conversion. The CapSense Component allows the following configurations of the IDACs:

- Enabling or disabling of Compensation IDAC
- Enabling or disabling of Auto-calibration for the IDACs
- DAC code selection for Modulator and Compensation IDACs if auto-calibration is disabled

Compensation IDAC

Enabling the compensation IDAC is called “dual IDAC” mode, and results in increased signal as explained in [Conversion gain in dual IDAC mode](#). Enable the compensation IDAC for most cases. Disable the compensation IDAC only if you want to free the IDAC for other general-purpose analog functions.

Auto-calibration

This feature enables the firmware to automatically calibrate the IDAC to achieve the required calibration target of 85%. It is recommended to enable auto-calibration for most cases. Enabling this feature will result in the following:

- Fixed raw count calibration to 85% of maximum raw count even with part-to-part C_P variation
- Avoids [Flat-spots](#)
- Automatically selects the optimum gain

If your design environment includes large temperature variation, you may find that the 85% IDAC calibration level is too high, and that the raw counts saturate easily over large changes in temperature, leading to lower SNR. If this is the case, you can adjust the calibration level lower by using `CapSense_CSDCalibrateWidget()` in your firmware.

For proper functioning of CAPSENSE™ under diverse environmental conditions, it is recommended to avoid very low or high IDAC codes. For a 7-bit IDAC, it is recommended to use IDAC codes between 18-110 from the possible 0 to 127 range. You can use CAPSENSE™ tuner to confirm that the auto-calibrated IDAC values fall in this recommended range. If the IDAC values are out of the recommended range, based on [Equation 25](#), [Equation 26](#) and [Equation 28](#), you may change the V_{Ref} or F_{SW} to get the IDAC code in proper range.

Disable IDAC auto-calibration if a change in C_P needs to be detected by measuring the raw count level at reset, for example:

- Detecting large variations in sensor C_P across boards or layout problems
- Detecting finger touch at reset
- Advanced CAPSENSE™ methods like liquid-level sensing, for example, to have different raw count level for different liquid levels at reset

Selecting DAC codes

This is not the recommended approach. However, this could be used only if you want to disable auto-calibration for any reason. To get the IDAC code, you may first configure CAPSENSE™ Component with auto-calibration enabled and all other hardware parameters the same as required for final tuning and read back the calibrated IDAC values using [Tuner GUI](#). Then, re-configure the CAPSENSE™ Component to disable auto-calibration and use the obtained IDAC codes as fixed DAC codes read-back from the [Tuner GUI](#).

5 PSoC™ 6 application notes

~~Scan resolution~~

It governs the sensor scan time per [Equation 29](#) and the conversion gain per [Equation 25](#), [Equation 25](#), and [Equation 27](#). Scan resolution needs to be selected to maintain a balance between the signal and scan time.

Higher scan resolution implies the following:

- Longer scan time per [Equation 29](#)
- Higher SNR on raw counts (increase in resolution increases the signal at a disproportionate rate to noise)

In general, it is recommended to tune the resolution to achieve as high SNR as possible; however if the system is constrained on power consumption and/or response time, set the lowest resolution to achieve at-least 5:1 SNR in the end system.

Note: *You should tune the scan resolution for less than 10:1 SNR only if you have scan time or power number constraints.*

Tuning shield electrode

The shield related parameters need to be additionally configured or tuned differently when you enable the Shield electrode in the CSD sensing method for liquid tolerance or reducing the C_p of the sensor.

Shield electrode tuning theory

Ideally, the shield waveform should be exactly the same as that of the sensor as explained in [Driven shield signal and shield electrode](#). However, in practical applications, the shield waveform may have a higher settling time and an overshoot error. Observe the sensor and shield waveform in the oscilloscope; an example waveform is shown in [Driven shield signal and shield electrode](#). The shield waveform should settle to the sensor voltage within 90% of ON time of the sense clock waveform and the overshoot error of the shield signal with respect to V_{REF} should be less than 10%.

If these conditions are not satisfied, you will observe a change in raw count of the sensors when touching the shield hatch; in addition, if inactive sensors are connected to shield as mentioned in [Inactive sensor connection](#), touching one sensor can cause change in raw count on other sensors, which indicates that there is cross talk if the shield electrode is not tuned properly.

In SmartSense, the sense clock frequency is automatically set. Check if these conditions are satisfied. If not satisfied, switch to [Manual tuning](#) and set the Sense clock frequency manually so that these conditions are satisfied. You can also tune the [Shield SW resistance](#) parameter to reduce the overshoot error.

5 PSoC™ 6 application notes

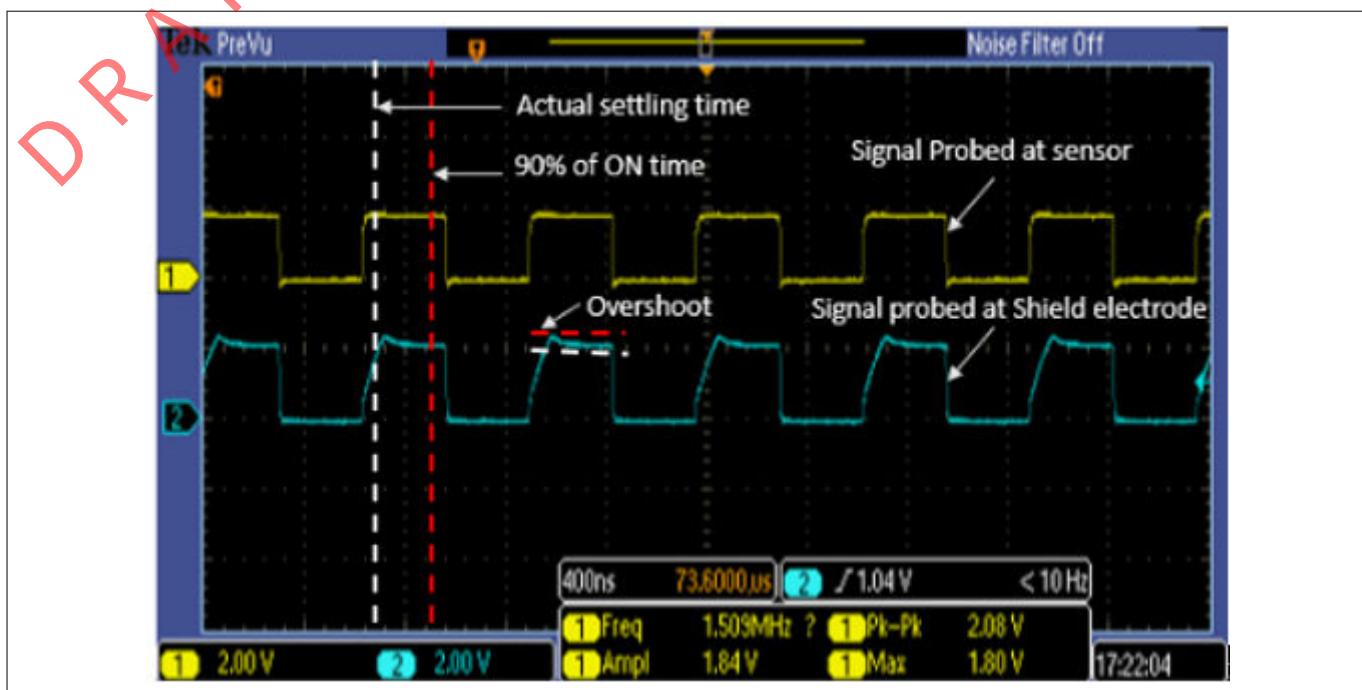


Figure 254 Properly tuned shield waveform

Tuning shield-related parameters

Enable shield tank capacitor

Enabling a shield tank capacitor increases the drive strength of the shield thus allowing the shield signal to settle to the sensor voltage faster as required. It is recommended to use the shield tank capacitor for PSoC™ 4A-S and PSoC™ 6 MCU family of devices. For PSoC™ 4A, PSoC™ 4A-L, and PSoC™ 4A-M family of devices, the shield tank capacitor does not prove very advantageous because it doubles the shield series resistance. It is recommended to keep this option disabled for these device families.

Shield electrode delay

For proper operation of the shield electrode, the shield signal should match the sensor signal in phase. Due to the difference in trace lengths of the sensor and shield electrodes, the shield waveform may arrive earlier to the sensor waveform. You can use an oscilloscope to view both sensor and shield signals to verify this condition. If they are not aligned, use this option to add delay to the shield signal to align the two signals. Available delays vary depending on the device selected.

Shield SW resistance

This parameter controls the shield signal rise and fall times to reduce EMI. This parameter is valid only for PSoC™ 4 S-Series, PSoC™ 4100S Plus, PSoC™ 4100PS, and PSoC™ 6 MCU family of devices. The default value of shield switch resistance is Medium. [Table 43](#) shows the effect of the Shield SW resistance value. You should select this value based on the application requirement; in addition, ensure that it satisfies the conditions in [Shield electrode tuning theory](#).

~~5 PSoC™ 6 application notes~~

Table 43 Shield SW resistance selection guidelines

Lower switch resistance	Higher switch resistance
Large overshoot error	Smaller overshoot error
Higher electromagnetic emission	Lower electromagnetic emission
Faster settling time that is, higher maximum sense clock frequency	Slower settling time that is, lower maximum sense clock frequency

Number of shield electrodes

This parameter specifies the number of shield electrodes required in the design. Most designs work with one dedicated shield electrode; however, some designs require multiple dedicated shield electrodes for ease of PCB layout routing or to minimize the PCB real estate used for the shield layer. See [Layout guidelines for shield electrode](#).

Inactive sensor connection

When the shield electrode is enabled for liquid-tolerant designs, or if you want to use shield to reduce the sensor parasitic capacitance, this option should be specified as “Shield”; otherwise, select “Ground”.

However, there is a risk of higher radiated emission due to inactive sensors getting connected to Shield. In such situations, use the CAPSENSE™ API to manually control inactive sensor connections. Instead of connecting all unused sensors to the shield, connect only the opposing inactive sensors or inactive sensors closer to the sensor being scanned to shield for reducing the radiated emission.

Selecting CAPSENSE™ software parameters

CAPSENSE™ software parameters govern the sensor status based on the raw count of a sensor. [Table 44](#) provides a list of CAPSENSE™ software parameters. These parameters apply to both CSD and CSX sensing methods. This section defines these parameters with the help of [Baseline](#), and provides guidance on how to adjust these parameters for optimal performance of your design. [Table 45](#) shows the recommended values for the software threshold parameter and they are applicable for most of the designs. However, if there are any external noise present in the end system, you must modify these thresholds accordingly to avoid any sensor false trigger.

Table 44 CAPSENSE™ component widget threshold parameters

Sl. No.	CAPSENSE™ component parameter name in PSoC™ Creator/ModusToolbox™
1.	Finger threshold
2.	Noise threshold
3.	Hysteresis
4.	ON debounce
5.	Sensor auto-reset
6.	Low baseline reset
7.	Negative noise threshold

5 PSoC™ 6 application notes

DRAFT Table 45 Recommended values for the threshold parameters

Sl. No.	CAPSENSE™ threshold parameter	Recommended value
1.	Finger threshold	80 percent of signal
2.	Noise threshold	40 percent of signal
3.	Hysteresis	10 percent of signal
4.	ON debounce	3
5.	Low baseline reset	30
6.	Negative noise threshold	40 percent of signal

Baseline

After tuning the CAPSENSE™ Component for a given C_p , the raw count value of a sensor may vary gradually due to changes in the environment such as temperature and humidity. Therefore, the CAPSENSE™ Component creates a new count value known as baseline by low-pass filtering the raw counts. **Baseline** keeps track of, and compensates for, the gradual changes in raw count. The baseline is less sensitive to sudden changes in the raw count caused by a touch. Therefore, the baseline value provides a reference level for computing signal.

Figure 255 shows the concept of raw count, baseline, and signal.

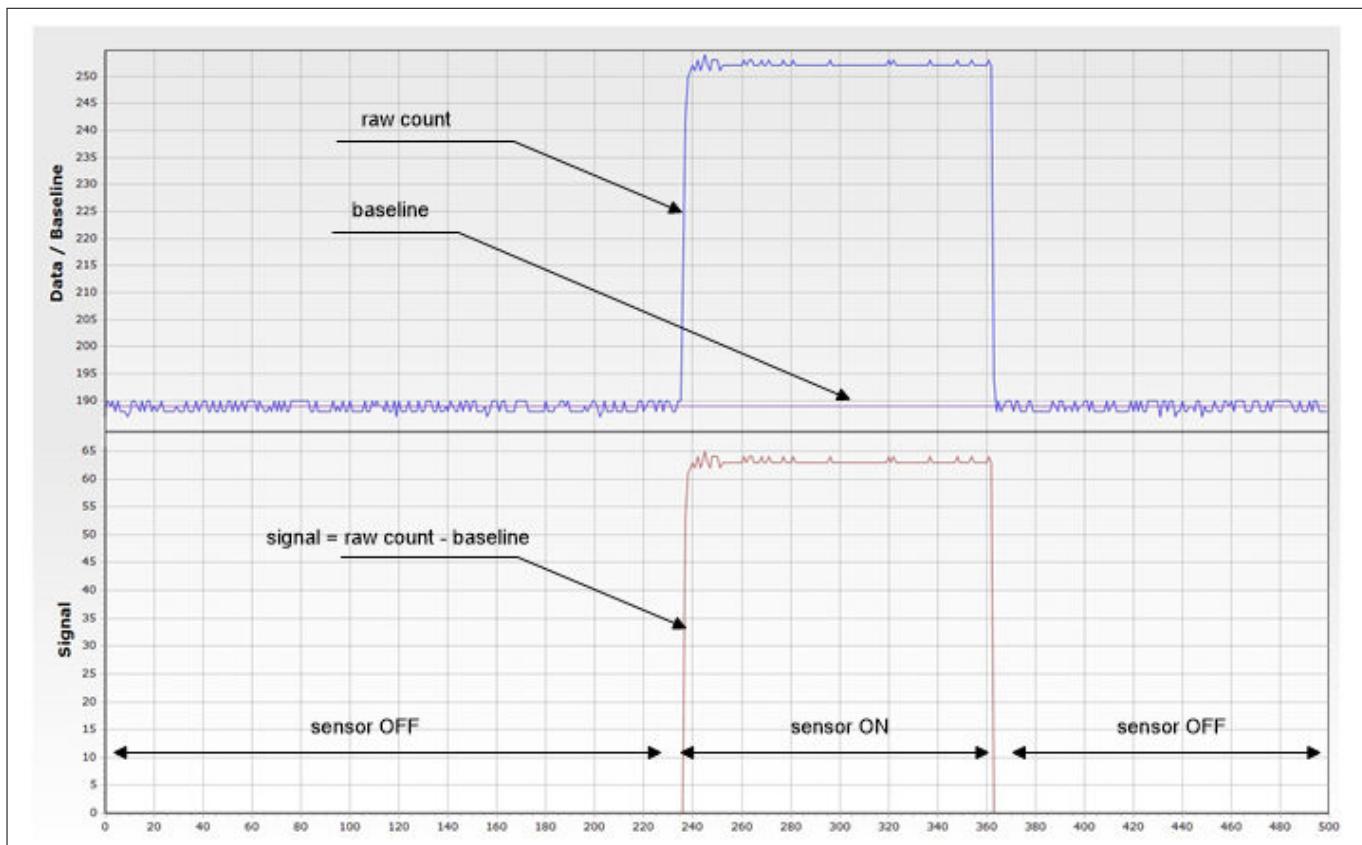


Figure 255 Raw count, baseline, and signal

Baseline update algorithm

To properly tune the CAPSENSE™ software, that is, the threshold parameters, it is important to understand how baseline is calculated and how the threshold parameters affect the baseline update.

5 PSoC™ 6 application notes

~~DRAFT~~

Baseline is a low-pass-filtered version of raw counts. As Figure 256 shows, baseline is updated by low-pass-filtering raw counts if the current raw count is within a range of (*Baseline* – *Negative noise threshold*) to (*Baseline* + *Noise threshold*). If the current raw count is higher than baseline by a value greater than noise threshold, baseline remains at a constant value equal to prior baseline value.

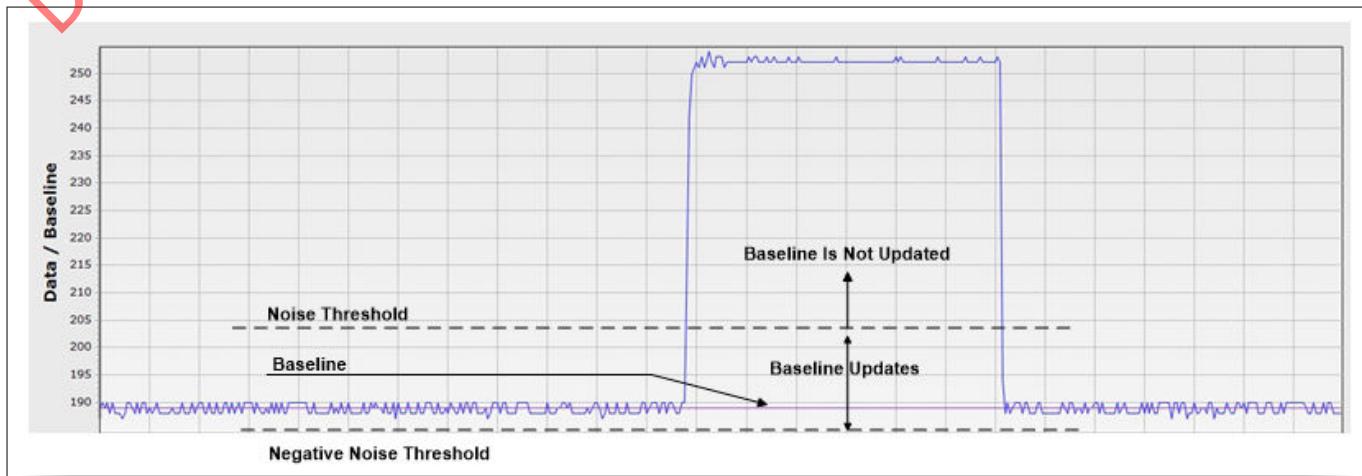


Figure 256 Baseline update algorithm

If the current raw count is below baseline minus negative noise threshold, baseline again remains constant at a value equal to prior baseline value for [Low baseline reset](#) number of sensor scans. If the raw count continuously remains lower than baseline minus noise threshold for low baseline reset number of scans, the baseline is reset to the current raw count value and starts getting updated again, as Figure 257 shows.

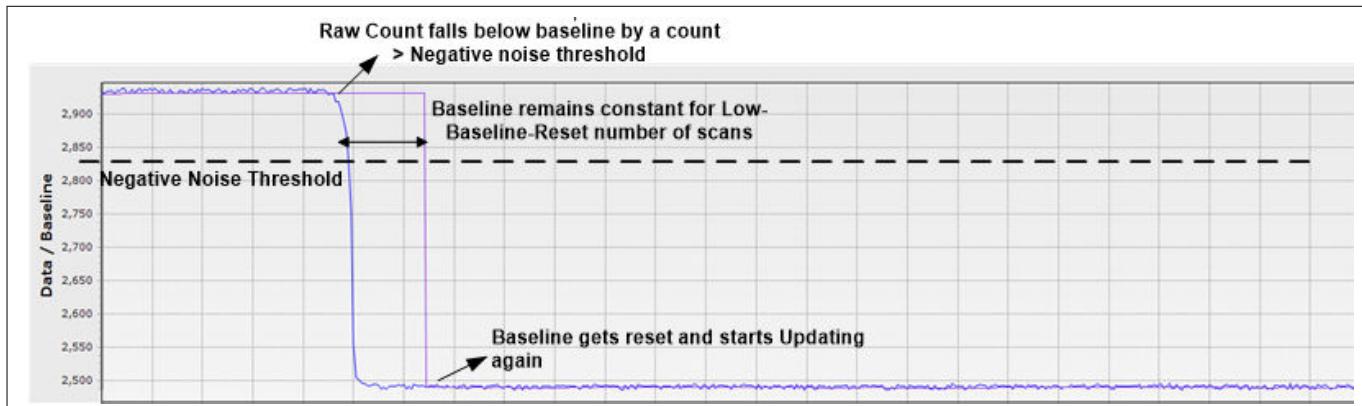


Figure 257 Low baseline reset

Finger threshold

The finger threshold parameter is used along with the hysteresis parameter to determine the sensor state, as [Equation 34](#) shows.

$$\text{Sensor state} = \begin{cases} \text{ON} & \text{if } (\text{Signal} \geq \text{Finger Threshold} + \text{Hysteresis}) \\ \text{OFF} & \text{if } (\text{Signal} \leq \text{Finger Threshold} - \text{Hysteresis}) \end{cases}$$

Equation 34 Sensor state

Note: *Signal* in the above equation refers to the difference: raw count – baseline, when the sensor is touched, as Figure 255 shows.

5 PSoC™ 6 application notes

~~DRAFT~~

It is recommended to set finger threshold to 80 percent of the signal. This setting allows enough margin to reliably detect sensor ON/OFF status over signal variations across multiple PCBs.

Hysteresis

The hysteresis parameter is used along with the finger threshold parameter to determine the sensor state, as [Equation 34](#) and [Figure 258](#) show. Hysteresis provides immunity against noisy transitions of sensor state. The hysteresis parameter setting must be lower than the finger threshold parameter setting. It is recommended to set hysteresis to 10 percent of the signal.

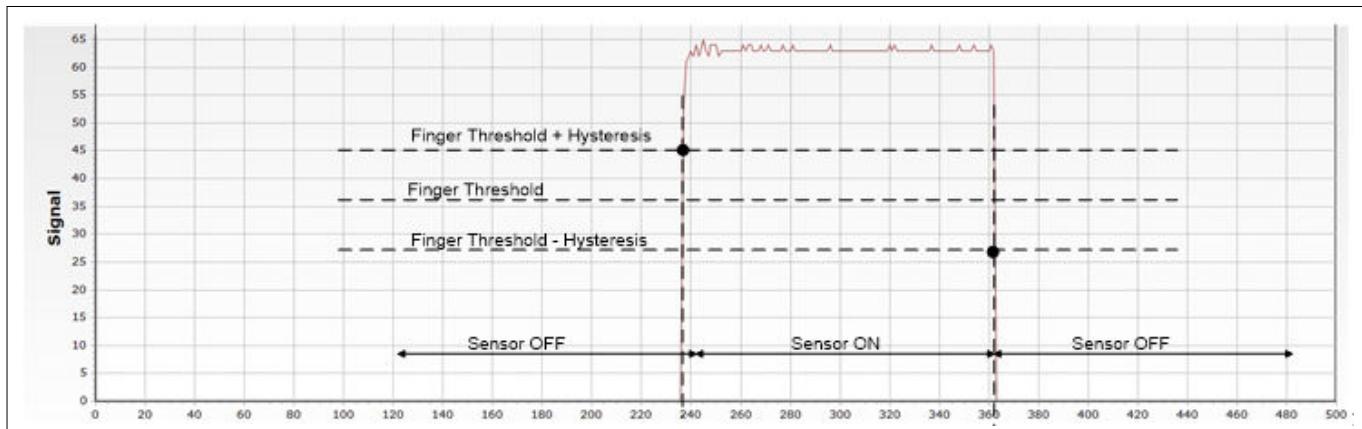


Figure 258 **Hysteresis**

Noise threshold

For single-sensor widgets, such as buttons and proximity sensors, the noise threshold parameter sets the raw count limit above which the baseline is not updated, as [Figure 256](#) shows. In other words, the baseline remains constant as long as the raw count is above *baseline + noise threshold*. This prevents the baseline from following raw counts during a finger touch.

The noise threshold value should always be lower than the *finger threshold - hysteresis*. It is recommended to set noise threshold to 40 percent of the signal.

If the noise threshold is set to a low value, the baseline will remain constant if raw counts suddenly increase by a small amount, say because of small shifts in power supply or shifts in ground voltage because of high GPIO sink current and so on.

On the other hand, if the noise threshold is set to a value close to *finger threshold - hysteresis*, the baseline may keep updating even when the sensor is touched. This will lead to reduced signal

Note: $(\text{signal} = \text{raw count} - \text{baseline})$ and the sensor state may not be reported as ON.

Negative noise threshold

The negative noise threshold parameter sets the raw count limit below which the baseline is not updated for the number of samples specified by the low baseline reset parameter as [Figure 257](#) shows.

Negative noise threshold ensures that the baseline does not fall low because of any high amplitude repeated negative noise spikes on raw count caused by different noise sources such as electrostatic discharge (ESD) events.

It is recommended to set the negative noise threshold parameter value to be equal to the noise threshold parameter value.

~~5 PSoC™ 6 application notes~~

~~Low baseline reset~~

This parameter is used along with the negative noise threshold parameter. It counts the number of abnormally low raw counts required to reset the baseline as [Figure 257](#) shows.

If a finger is placed on the sensor during device startup, the baseline is initialized to the high raw count value at startup. When the finger is removed, raw counts fall to a lower value. In this case, the baseline should track the low raw counts. The Low Baseline Reset parameter helps to handle this event. It resets the baseline to the low raw count value when the number of low samples reaches the low baseline reset number.

Note: *In this case, when the finger is removed from the sensor, the sensor will not respond to finger touches for a low baseline reset time given by [Equation 35](#).*

$$\text{Low Baseline Reset Time} = \frac{\text{Low Baseline Re set parameter value}}{\text{Scan rate}}$$

Equation 35 Low baseline reset time

The low baseline reset parameter should be set to meet following conditions:

- Low baseline reset time is greater than the time for which negative noise (due to noise sources such as ESD events) is expected to last
- Low baseline reset time is lower than the time in which a sensor is expected to start responding again after the finger kept on sensor during device startup is removed from the sensor

The low baseline reset parameter is generally set to a value of 30.

Debounce

This parameter selects the number of consecutive CAPSENSE™ scans during which a sensor must be active to generate an ON state from the component. Debounce ensures that high-frequency, high-amplitude noise does not cause false detection.

$$\begin{aligned} \text{Sensor state} = & \{ \text{ON if } (\text{Signal} \geq \text{Finger Threshold} + \text{Hysteresis}) \text{ for scans} \geq \text{debounce} \\ & \text{OFF if } (\text{Signal} \leq \text{Finger Threshold} - \text{Hysteresis}) \\ & \text{OFF if } (\text{Signal} \geq \text{Finger Threshold} + \text{Hysteresis}) \text{ for scans} < \text{debounce} \} \end{aligned}$$

Equation 36 Sensor state with debounce

The Debounce parameter impacts the response time of a CAPSENSE™ system. The time it takes for a sensor to report ON after the raw counts value have increased above finger threshold + hysteresis because of finger presence, is given by [Equation 37](#).

$$\text{Sensor response time} = \frac{\text{Debounce}}{\text{Scan Rate}}$$

Equation 37 Relationship between debounce and sensor response time

The Debounce parameter is generally set to a value of ‘3’ for reliable sensor status detection. It can be raised or lowered based on the noise aspects of the end user system.

Sensor auto reset

Enabling the Sensor Auto Reset parameter causes the baseline to always update regardless of whether the signal is above or below the noise threshold.

5 PSoC™ 6 application notes

~~DRAFT~~

When auto reset is disabled, the baseline only updates if the current raw count is within a range of (*Baseline* - *Negative Noise Threshold*) to (*Baseline* + *Noise Threshold*) as Figure 256 shows and the [Baseline update algorithm](#) describes. However, when Auto Reset is enabled, baseline is always updated if the current raw count is higher than (*Baseline* - *Negative Noise Threshold*) as Figure 259 shows.

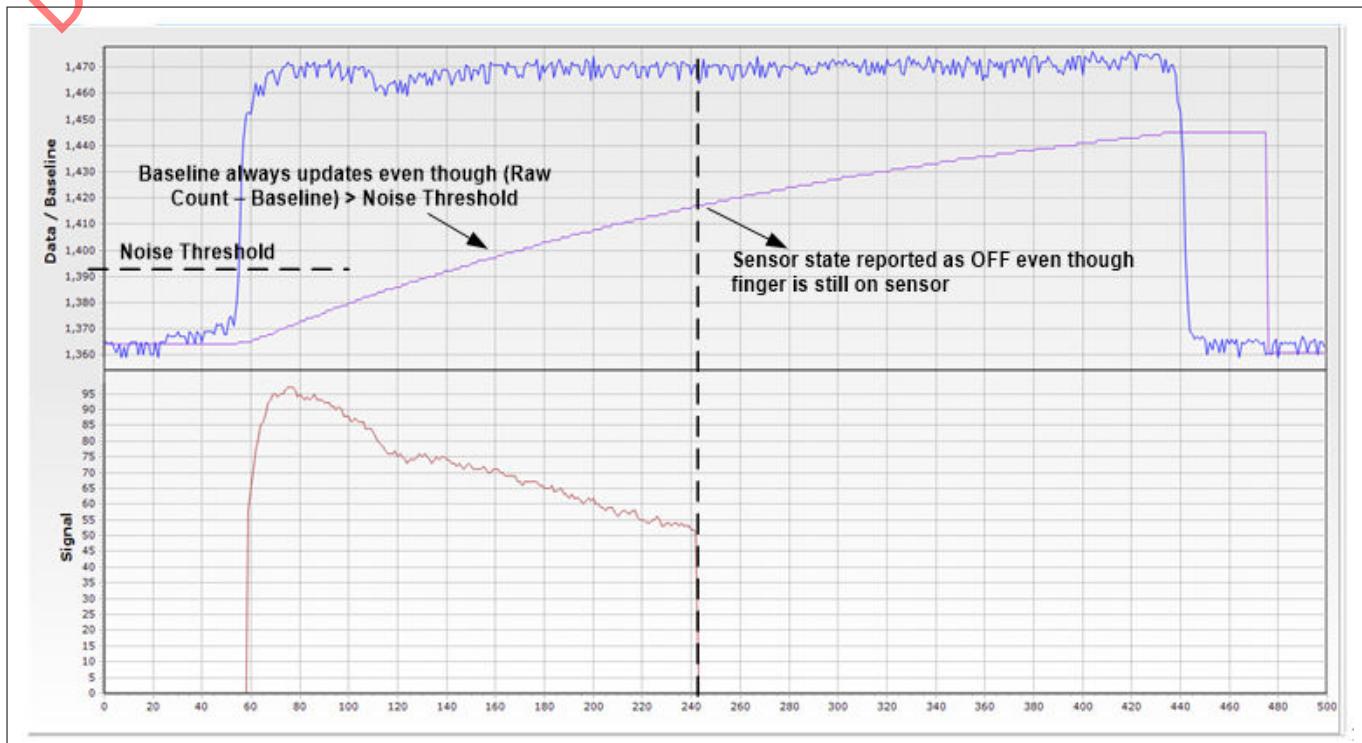


Figure 259 Baseline update with sensor auto reset enabled

Because the baseline is always updated when sensor auto reset is enabled, this setting limits the maximum time duration for which the sensor will be reported as pressed. However, enabling this parameter prevents the sensors from permanently turning on if the raw count suddenly rises without anything touching the sensor. This sudden rise can be caused by a large power supply voltage fluctuation, a high-energy RF noise source, or a very quick temperature change.

Enable this option if you have a problem with sensors permanently turning on when the raw count suddenly rises without anything touching the sensor.

Multi-frequency scan

Enabling multi-frequency scan, the CAPSENSE™ component performs a sensor scan with three different sense clock frequencies and obtains corresponding difference count. The median of the sensor difference-count is selected for further processing. Use this feature for robust operation in the presence of external noise at a certain sensor scan frequency. This option is not available in SmartSense Full Autotune mode. See the code example [CE227719 CAPSENSE™ with multi-frequency scan](#).

Button widget tuning

Figure 260 illustrates an overview of the CSD button tuning procedure.

5 PSoC™ 6 application notes

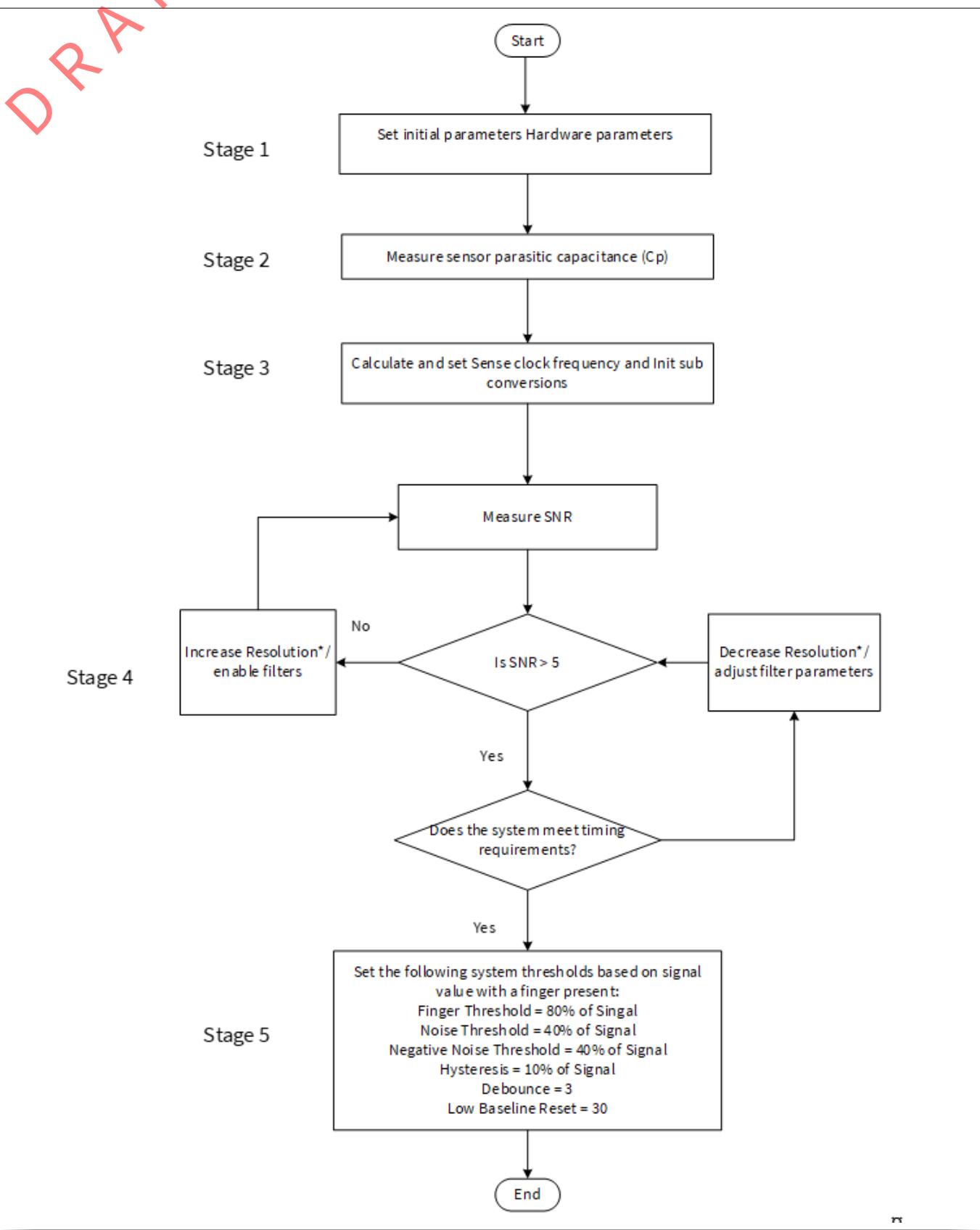


Figure 260 CSD button widget tuning flowchart

Attention: For fifth-generation CAPSENSE™, change number of sub-conversions (N_{Sub}) instead of resolution.

~~5 PSoC™ 6 application notes~~

To review the hardware design, see the [Sensor construction](#) and [PCB layout guidelines](#) sections in the [Design considerations](#) chapter. Also, see the [Tuning debug FAQs](#) section for guidelines on advanced debug.

As explained in [Chapter 5.8.5.1](#), Manual tuning requires effort to tune optimum CAPSENSE™ parameters, but allows strict control over characteristics of capacitive sensing system, such as response time and power consumption. The button is tuned for reliable touch detection to avoid false triggers in noisy environment.

The [CE230926 PSoC™ 4: CAPSENSE™ CSD button tuning](#) explains tuning of self-capacitance based button widgets in the Eclipse IDE for ModusToolbox™ using the CAPSENSE™ Tuner GUI. For details on the Component and all related parameters, see the [Component datasheet](#).

Slider widget tuning

A slider has many segments, each of which is connected to the CAPSENSE™ input pins of the PSoC™ device. Unlike the simple on/off operation of a button widget sensor, slider widget sensors work together to track the location of a finger or other conductive object. Because of this, the slider layout design should ensure that the C_p of all the segments in a slider remain as close as possible. Keeping similar C_p values between sensors will help minimize the tuning effort and ensure an even response across the entire slider. See [Slider design](#) for details on slider layout design guidelines to avoid nonlinearity in the centroid, ensure that the signal from all the slider segments is equal, as [Figure 261](#) shows, when a finger is placed at the center of the slider segment. If the signal of the slider segments is different, then the centroid will be nonlinear, as [Figure 262](#) shows.

Note: *In PSoC™ Creator and in ModusToolbox™, a centroid of 0xFFFF and 0x0000 is reported respectively when a finger is not detected on the slider, or when none of the slider segments report a difference count value greater than the Finger Threshold parameter.*

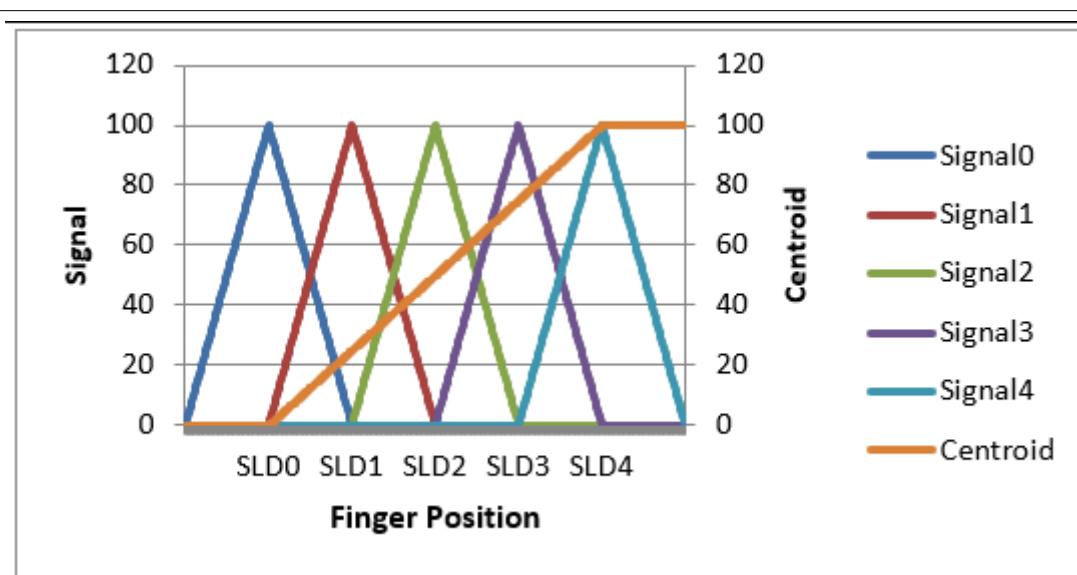


Figure 261 Response of centroid versus finger location when signals of all slider elements are equal

Note: *Signal = Raw Count – Baseline*

5 PSoC™ 6 application notes

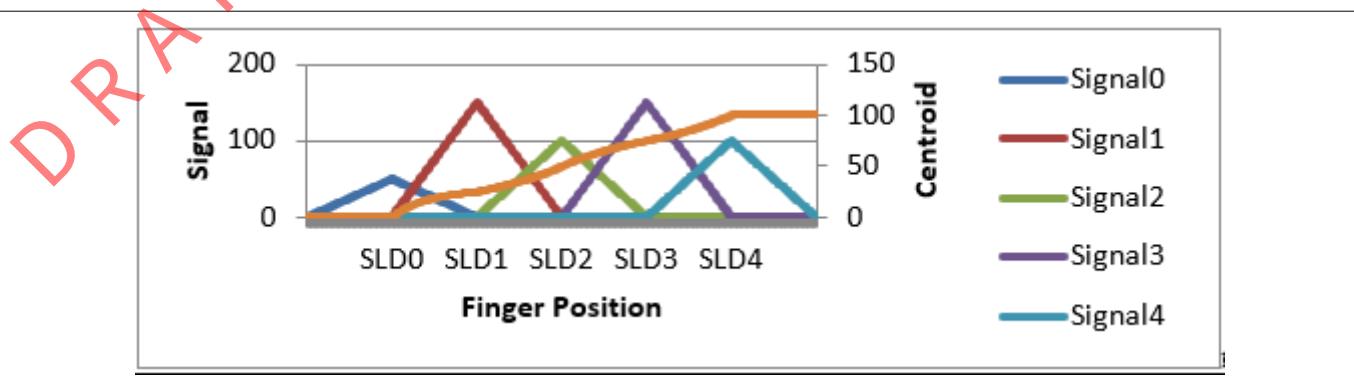


Figure 262 Response of centroid versus finger location when the signal of all slider elements are different

A linear response for the reported finger position (that is, the Centroid position) versus the actual finger position on a slider requires that the slider design is such that whenever a finger is placed anywhere between the middle of the segment SLD n and middle of segment SLD $n-1$, other than the exact middle of slider segments, exactly two sensors report a valid signal ²⁰⁾. If a finger is placed at the exact middle of any slider segment, the adjacent sensors should report a difference count = noise threshold. These conditions are required since the centroid position calculation is based on the closest segment to the finger and two neighboring segments as shown in [Equation 38](#).

$$\text{centroid position} = \left(\frac{S_{x+1} - S_{x-1}}{S_{x+1} + S_{x-1}} + x \right) \times \frac{\text{Resolution}}{(n - 1)}$$

Equation 38 Centroid algorithm used by CAPSENSE™ component in PSoC™

Where,

Resolution = API resolution set in the CAPSENSE™ Component Customizer

n = Number of sensor elements in the CAPSENSE™ Component Customizer

x = Index of element which gives maximum signal

s_i = Different counts (with subtracted noise threshold value) of the slider segment

[Figure 263](#) shows an overview of the CSD slider tuning procedure.

²⁰⁾ Here, a valid signal means that the difference count of the given slider segment is greater than or equal to the noise threshold value.

5 PSoC™ 6 application notes

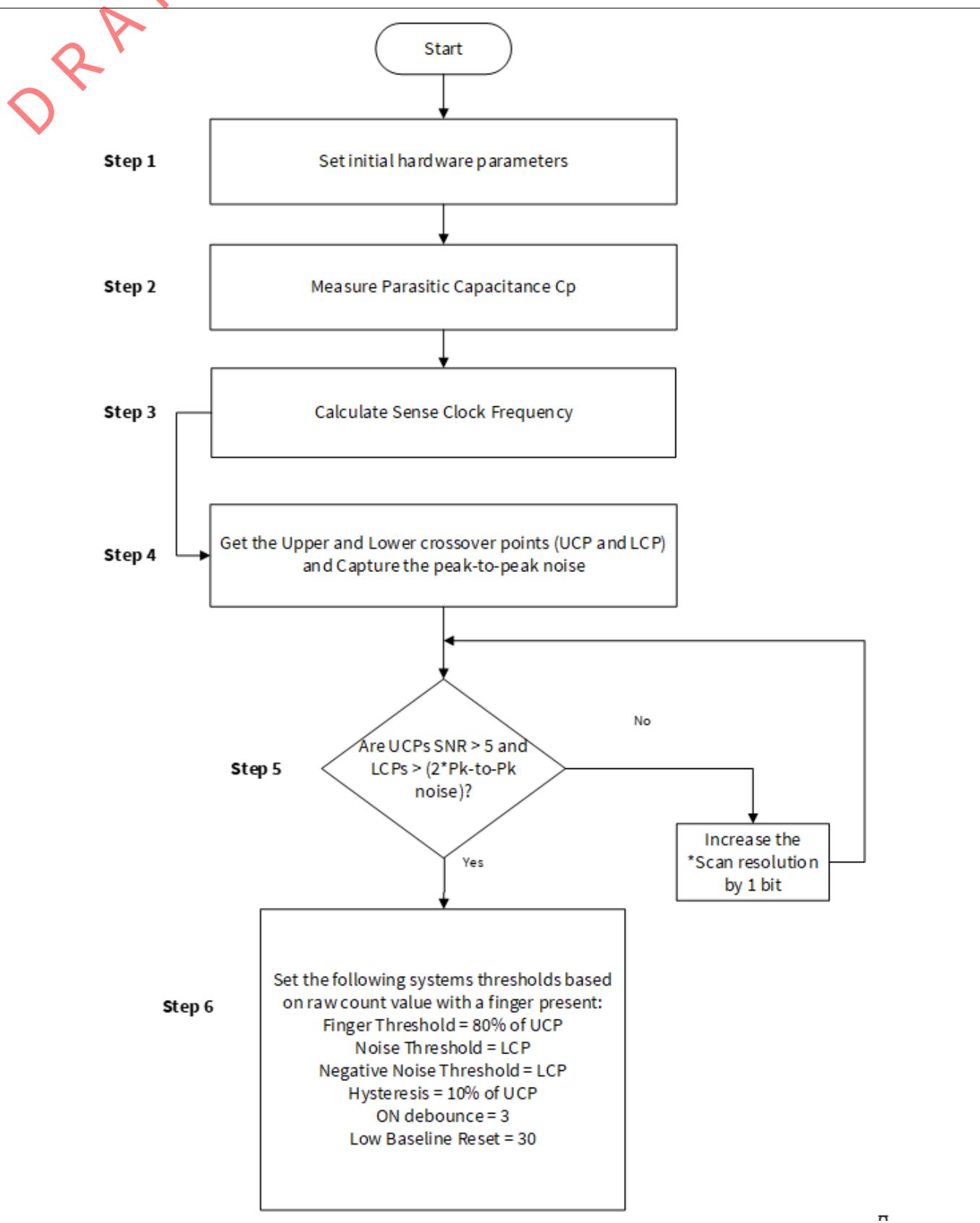


Figure 263 CSD slider widget tuning flowchart

Attention: For fifth-generation CAPSENSE™, change number of sub-conversions (N_{Sub}) instead of resolution.

~~5 PSoC™ 6 application notes~~

The upper crossover point (UCP) and lower crossover point (LCP) are obtained as shown in [Figure 264](#). Refer to [CE229521 – PSoC™ 4 CAPSENSE™ CSD Slider tuning](#) which demonstrates how to manually tune a self-capacitance based Slider widget on PSoC™ Creator and [CE230493 – PSoC™ 4: CAPSENSE™ CSD Slider tuning](#) on Eclipse IDE for ModusToolbox™.

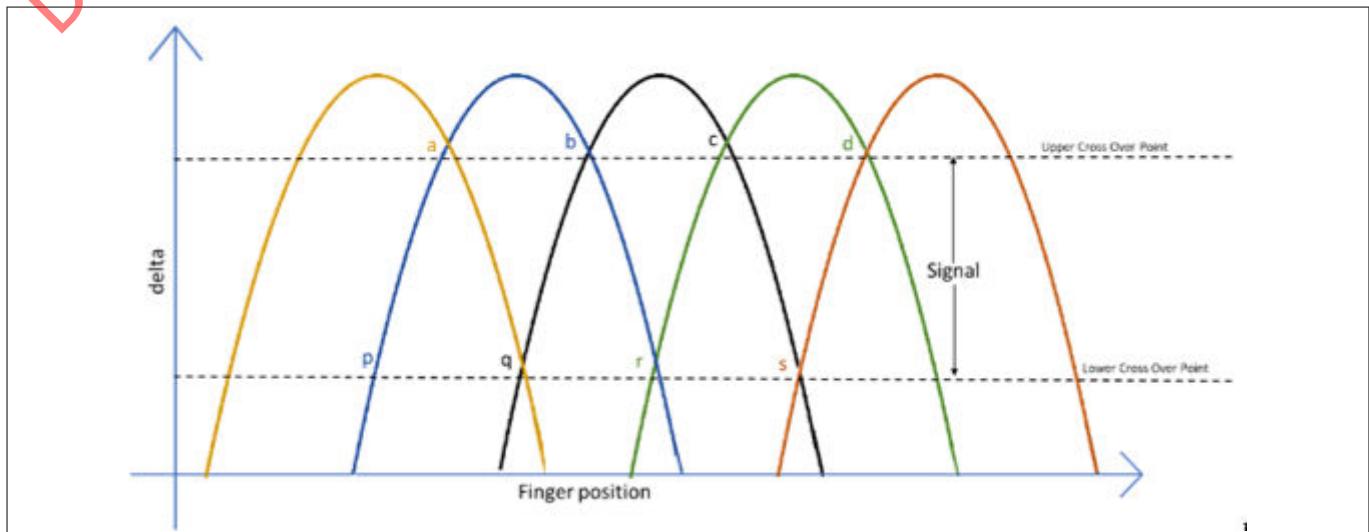


Figure 264 **Difference count (delta) vs finger position**

Touchpad widget tuning

A self-capacitance-based touchpad is essentially two sliders implemented in the horizontal and vertical directions. Hence, it is also tuned in a similar way as that of a slider, to obtain an even response across the trackpad/touchpad. To gain true multi-touch performance, it is recommended to use mutual-capacitance based touchpad. The centroid algorithm obtains the signals (diff-counts) from all the segments and calculates the x and y position co-ordinates.

The CSD Touchpad reuses Slider's centroid algorithm that is applied individually to row and column sensors treated as simple sliders. Hence, the centroid position calculation formula for CSD Touchpad is same as [Equation 38](#).

CSD finger detection criteria

The touch in a CSD Touchpad is reported to the host when the following Finger detection criteria is satisfied:

1. $Z_{peak} > (FingerThreshold \pm Hysteresis)$
2. $Z_{peak} > (FingerThreshold \pm Hysteresis) \times \frac{Z3_Filt_Scale}{2} \rightarrow (\text{At panel edge})$
3. $Z_{Peak} > (Finger Threshold \pm Hysteresis) \times \frac{Z3_Filt_Scale}{4} \rightarrow (\text{At panel corner})$

Where,

Z_{Peak} = Maximum Signal when the finger is present at the centre of the sensor

$Z3_sum$ = Sum of Signals of segment with maximum signal and two neighboring segments

$Z3_Filt_Scale = (0.8 * Z3_Sum) / \text{Finger Threshold}$

The $Z3_Filt_Scale$ value ensures that the detected object is of the correct proportions.

$Z3_sum$ (of both row and column) condition is checked to see if the absolute mass of the finger is large enough to be recognized as a finger. The $Z3_sum$ condition may prevent noise-induced false touches.

5 PSoC™ 6 application notes

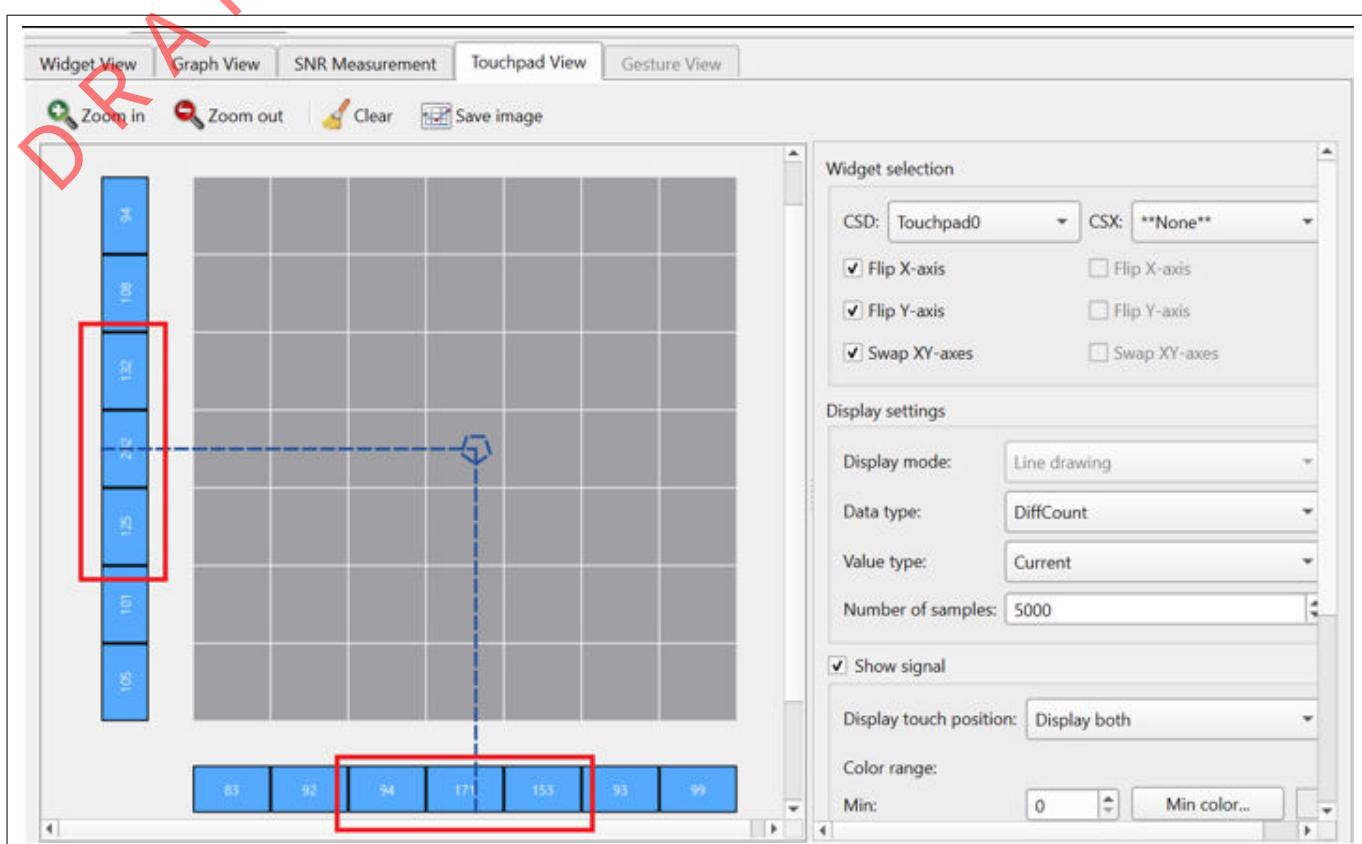


Figure 265 Z3_sum values on CAPSENSE™ tuner

Figure 266 shows an overview of the CSD touchpad tuning procedure.

5 PSoC™ 6 application notes

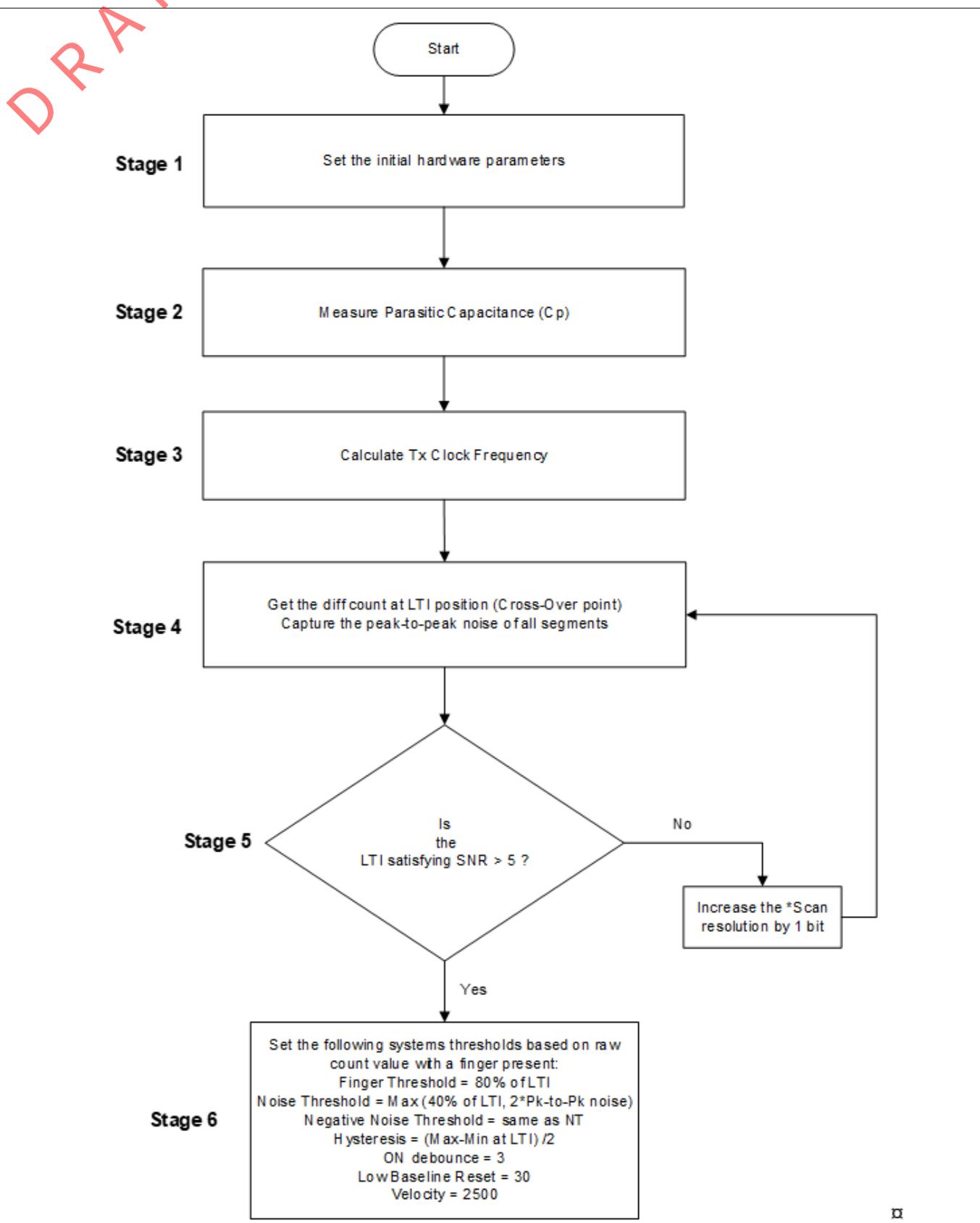


Figure 266 CSD touchpad widget tuning flowchart

Attention: For fifth-generation CAPSENSE™, change number of sub-conversions (N_{Sub}) instead of resolution.

~~5 PSoC™ 6 application notes~~

LTI measures the peak diff-count when a finger touch is centered between four sensors. The LTI signal count is the average of the four peak sensors. This gives the least valid touch signal.

Proximity widget tuning

For tuning a proximity sensor, see [AN92239 - Proximity sensing with CAPSENSE™](#).

5.8.5.3.3 CSX sensing method (third- and fourth-generation)

This chapter explains the basics of manual tuning using the CSX sensing method. It also explains the hardware parameters that influence a manual tuning procedure.

Basics

Conversion gain and CAPSENSE signal

In a mutual-capacitance sensing system, the $\text{Rawcount}_{\text{counter}}$ is directly proportional to the mutual-capacitance between the Tx and Rx electrodes, as [Equation 39](#) shows.

$$\text{Rawcount}_{\text{Counter}} = G_{\text{CSX}} C_M$$

Equation 39 Raw count relationship to sensor capacitance

Where,

G_{CSX} = Capacitance to digital conversion gain of CAPSENSE™ CSX

C_M = Mutual-capacitance between the Tx and Rx electrodes.

[Figure 268](#) shows the relationship between raw count and mutual capacitance of the CSX sensor. The tunable parameters of the conversion gain in [Equation 40](#) are F_{TX} , N_{MOD} , F_{MOD} and IDAC.

The approximate value of this conversion gain is:

$$G_{\text{CSX}} = \frac{2 V_{\text{TX}} F_{\text{TX}} \text{MaxCount}}{\text{IDAC}}$$

Equation 40 Capacitance to digital converter gain

$$\text{MaxCount} = \frac{F_{\text{Mod}} N_{\text{Sub}}}{F_{\text{TX}}}$$

Equation 41 MaxCount equation

Where,

V_{TX} = Voltage at the Tx node of the sensor as shown in [Figure 267](#)

$$V_{\text{TX}} = V_{\text{ON}} - V_{\text{OFF}}$$

The value of V_{TX} is always V_{DDIO} or V_{DDD} (if V_{DDIO} is not available) if the Tx clock frequency can completely charge and discharge the Tx electrode. F_{TX} is the Tx clock frequency, I_{DAC} is the current drawn for charging and discharging the C_{INT} capacitors, and N_{sub} is the number of sub-conversions.

5 PSoC™ 6 application notes

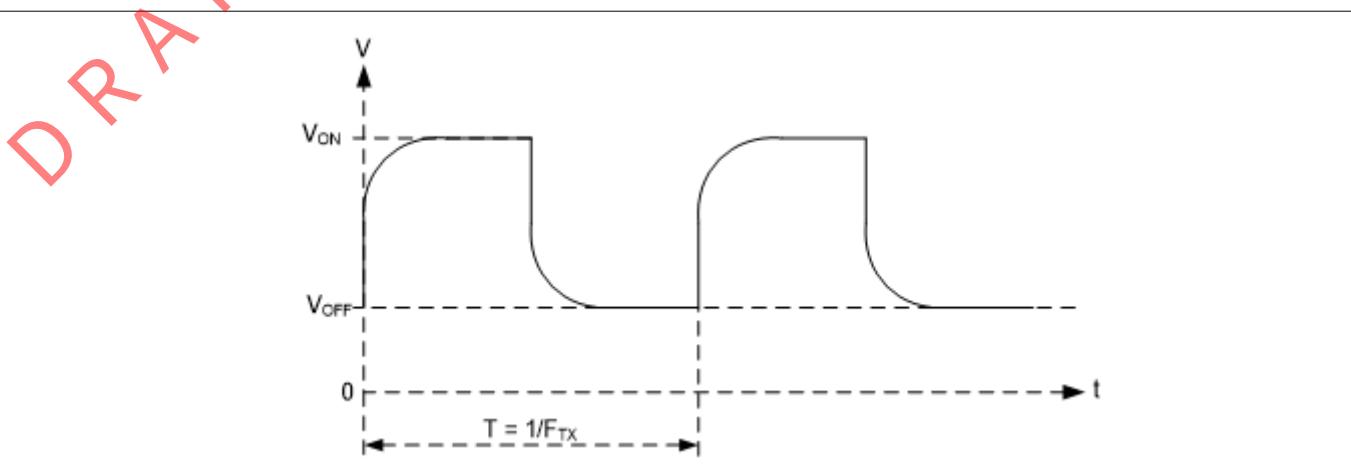


Figure 267 **Voltage at Tx node of the CSX sensor**

Note: The raw count observed from the Component is given by [Equation 42](#). See [CAPSENSE™ CSX sensing method \(third- and fourth- generation\)](#) for more details on $\text{Rawcount}_{\text{Component}}$.

$$\text{Rawcount}_{\text{Component}} = \text{MaxCount} - \text{Rawcount}_{\text{counter}}$$

Equation 42 **Rawcount_{Component}**

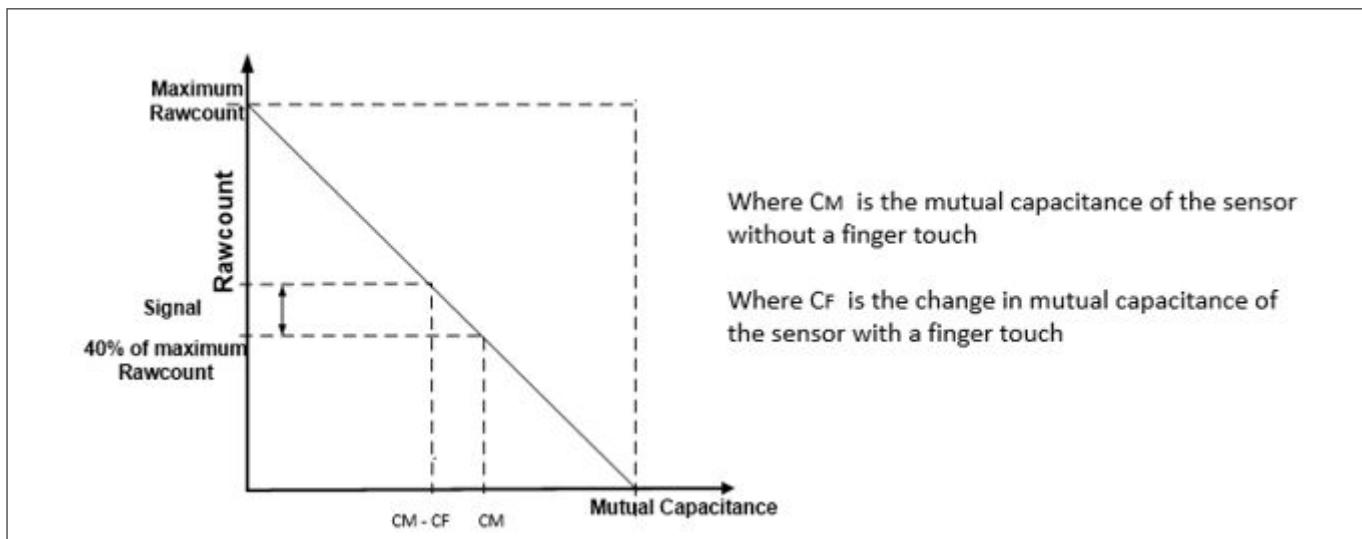


Figure 268 **Raw count vs Sensor mutual-capacitance**

Selecting CAPSENSE™ hardware parameters

CAPSENSE™ hardware parameters govern the conversion gain and. [Table 46](#) lists the CAPSENSE™ hardware parameters that apply to the CSX sensing method. [Table 46](#) also shows the mapping of each parameter in the PSoC™Creator CAPSENSE™ component to the one in the ModusToolbox™ middleware. For simplicity of documentation, this design guide shows selecting the CAPSENSE™ parameter using the CAPSENSE™ configurator in PSoC™ Creator. The same procedure could be followed in configuring CAPSENSE™ in ModusToolbox™. However, in ModusToolbox™, you set the Tx clock and Modulator clock using divider values. On the other hand, in PSoC™ Creator, you specify the frequency value directly in the configurator. See [Component datasheet/middleware document](#).

~~5 PSoC™ 6 application notes~~

Table 46 CAPSENSE™ signal component hardware parameters

Sl #	CAPSENSE™ parameter in PSoC™ Creator	CAPSENSE™ parameter in ModusToolbox™
1	Modulator clock frequency	Modulator clock divider
2	Tx clock source	Tx clock source
3	Tx clock frequency	Tx clock divider
4	IDAC	IDAC
5	Number of sub-conversions	Number of sub-conversions

Tx clock parameters

There are two parameters that are related to the Tx clock: Sense clock source and Sense clock frequency.

Tx clock source

Select “Auto” to let the Component automatically choose the best Tx clock source between Direct and Spread spectrum clock (SSCx) for each widget. If not selecting Auto, select the clock source based on the following:

- Direct – Clock signal with a fixed clock frequency. Use this option for most cases
- Spread spectrum clock (SSCx) – If you chose this option, the Tx clock signal frequency is dynamically spread over a predetermined range. Use this option for reduced EMI interference and avoiding Flat-spots

However, when selecting SSCx clock, you need to select the Tx clock frequency, Modulator clock frequency, and number of sub conversion such that the conditions mentioned in [Component datasheet/ModusToolbox™ CAPSENSE™ configurator guide](#) for SSCx clock source selection are satisfied.

Tx clock frequency

The Tx clock frequency determines the duration of each sub-conversion as explained in the [CAPSENSE™ CSX sensing method \(third- and fourth- generation\)](#) Chapter. The Tx clock signal must completely charge and discharge the sensor parasitic capacitance; it can be verified by checking the signal in an oscilloscope, or it can be set using the [Equation 43](#). In addition, you should ensure that the auto-calibrated IDAC code lies in the mid-range (for example, 30-90) for the selected . If the auto-calibrated IDAC code lies out of the recommended range, tune such that it IDAC falls in the recommended range and satisfies [Equation 43](#).

$$F_{TX} < \frac{1}{10 R_{SeriesTx} C_{PTx}}$$

Equation 43 Condition for selecting Tx clock frequency

To minimize the scan time, as [Equation 44](#) shows, it is recommended to use the maximum Tx clock frequency available in the component drop-down list that satisfies the criteria.

$$T_{CSX} = \frac{N_{Sub}}{F_{Tx}}$$

Equation 44 Scan time of CSX sensor

Where, N_{Sub} = [Number of sub-conversions](#).

Additionally, if you are using the SSCx clock source, ensure that you select the Tx clock frequency that meets the conditions mentioned in [Component datasheet/middleware document/ModusToolbox™ CAPSENSE™ configurator guide](#) in addition to these conditions.

5 PSoC™ 6 application notes

The maximum value of FTX depends on the selected device. For the PSoC™ 4 S-Series, PSoC™ 4100S Plus, PSoC™ 4100PS, and PSoC™ 6 MCU family of devices, the maximum FTX is 3000 kHz and for other devices it is 300 kHz.

Modulator clock frequency

It is best to choose the highest allowed clock frequency for the given device because a higher modulator clock frequency leads to a higher sensitivity/signal, increased accuracy, and lower noise for a given C_M to digital count conversion as [Equation 31](#) and [Equation 32](#) indicate. Also, a higher value of F_{mod}/F_{tx} ensures lower width of [Flat-spots](#) in C_M to raw count conversion.

IDAC

It is recommended to enable IDAC auto-calibration. It is best to avoid very high and very low IDAC codes. The recommended IDAC code range is between 30-90. If the IDAC values are away from the recommended range, tune the Tx clock frequency to adjust the IDAC level. If the IDAC is failing to calibrate properly, it may be due to low C_M in the design. Refer to the section [I am observing a low CM for my CSX button](#) for mitigating impact of low C_M in the design.

Number of sub-conversions

The number of sub-conversions decides the sensitivity of the sensor and sensor scan time. From [Equation 15](#) for a fixed modulator clock and Tx clock, increasing the number of sub-conversions (N_{Sub}) increases the signal and SNR. However, increasing the number of sub-conversions also increases the scan time of the sensor per [Equation 45](#).

$$\text{Scan time} = \frac{N_{Sub}}{F_{Tx}}$$

Equation 45 CSX scan time

Initially, set the value to a low number (for example, 20), and use the Tuner GUI to find the SNR of the sensor. If the SNR is not > 5:1 with the selected N_{Sub} , try to increase the N_{Sub} in steps such that the SNR requirement is met.

Selecting CAPSENSE™ software parameters

CAPSENSE™ software parameters for mutual-capacitance are the same as that for self-capacitance; therefore, these parameters could be selected as mentioned in the section [Selecting CAPSENSE™ software parameters](#)[Selecting CAPSENSE™ software parameters](#).

Button widget tuning

[Figure 269](#) illustrates an overview of the CSX button tuning procedure.

5 PSoC™ 6 application notes

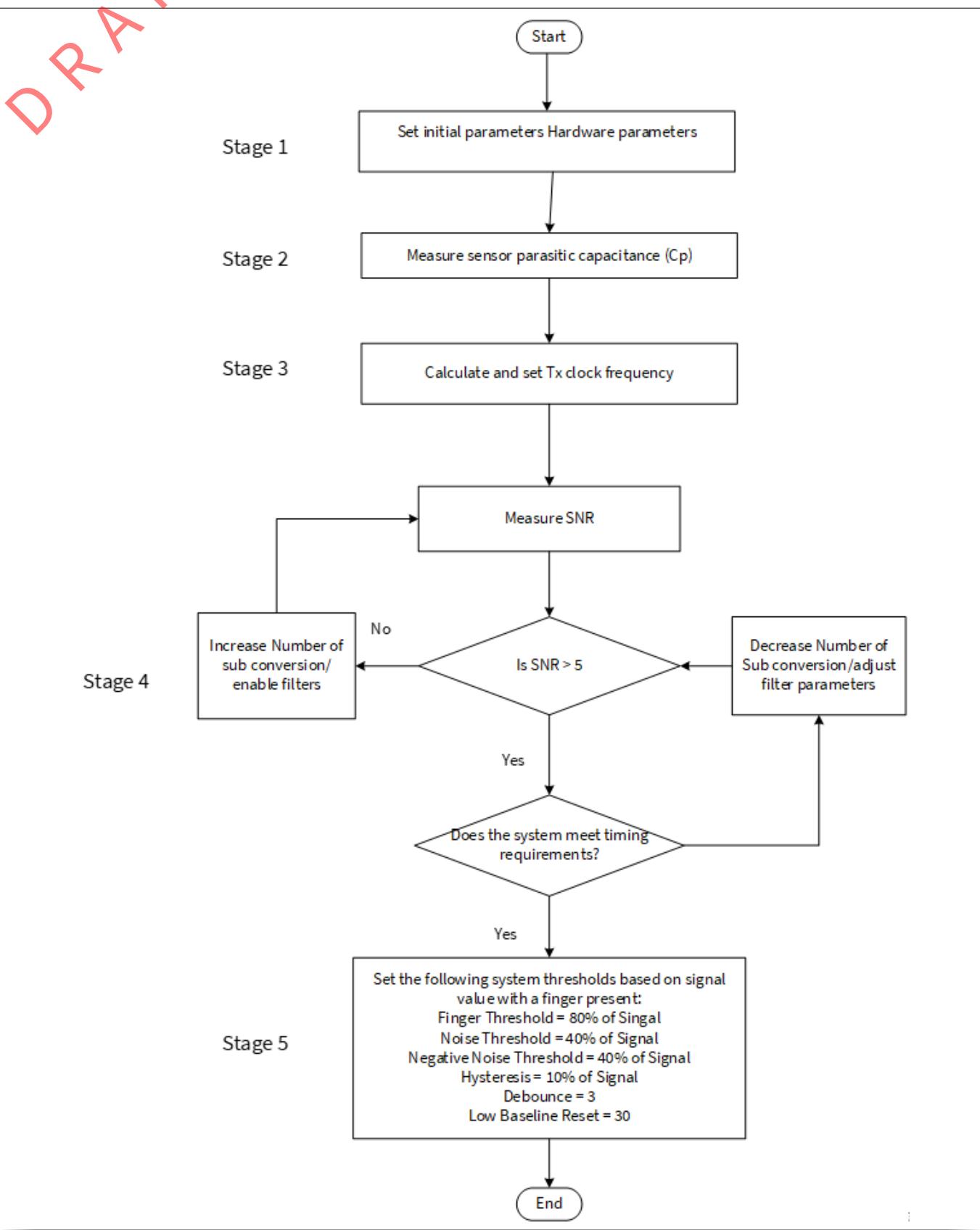


Figure 269 CSX button widget tuning example

* To review the hardware design, see the [Sensor construction](#) and [PCB layout guidelines](#) sections in the [Design considerations](#) chapter. Also, see the [Tuning debug FAQs](#) section for guidelines on advanced debug

~~5 PSoC™ 6 application notes~~

The [CE230660 PSoC™ 4: CAPSENSE™ CSX button tuning](#) explains tuning of mutual-capacitance based button widgets in the Eclipse IDE for ModusToolbox™ and [CE228931 – PSoC™ 4 CAPSENSE™ CSX button tuning](#) in PSoC™ Creator using the CAPSENSE™ tuner. For details on the Component and all related parameters, see the [Component datasheet](#).

Touchpad widget tuning

Mutual-capacitance based touchpad widget supports up to three simultaneous finger touches. A slightly different Centroid algorithm compared to CSD touchpad is applied in a CSX touchpad widget. A 3x3 algorithm is used for calculating the X and Y position using Centroid algorithm as shown in [Equation 46](#) and [Equation 47](#) respectively.

$$\text{positionX} = \left(\frac{S_{x+1} - S_{x-1}}{S_{3 \times 3}} + x \right) \times \frac{\text{ResolutionX}}{(n_x - 1)}$$

Equation 46 Calculating X-position using centroid algorithm in CSX touchpad

Where,

ResolutionX = Maximum X-axis position

n_x = Number of sensor elements in the X-direction

x = Index of element which gives maximum signal

S_{x+1} = Sum of three neighbor elements at the left from maximum (x)

S_{x-1} = Sum of three neighbor elements at the right from maximum (x)

$S_{3 \times 3}$ = Total sum of 3x3 difference array

$$\text{positionY} = \left(\frac{S_{y+1} - S_{y-1}}{S_{3 \times 3}} + y \right) \times \frac{\text{ResolutionY}}{(n_y - 1)}$$

Equation 47 Calculating Y-position using centroid algorithm in CSX touchpad

Where,

ResolutionY = Maximum Y-axis position

n_y = Number of sensor elements in the Y-direction

y = Index of element which gives maximum signal

S_{y+1} = Sum of three neighbor elements at the top from maximum (y)

S_{y-1} = Sum of three neighbor elements at the bottom from maximum (y)

CSX finger detection criteria

The touch in a CSX touchpad is reported to the host when the following Finger detection criteria is satisfied:

1. $Z_{\text{Peak}} > \text{Finger threshold} \pm \text{Hysteresis}$
2. $Z9_{\text{Sum}}$ condition
 - $Z9_{\text{Sum}} > ((\text{Finger threshold} + \text{Hysteresis}) * Z9_{\text{Filt_Scale}})$ (At panel core)
 - $Z9_{\text{Sum}} > ((\text{Finger threshold} + \text{Hysteresis}) * Z9_{\text{Filt_Scale}}/2)$ (At panel edge)
 - $Z9_{\text{Sum}} > ((\text{Finger threshold} + \text{Hysteresis}) * Z9_{\text{Filt_Scale}}/4)$ (At panel corner)
3. $Z8_{\text{sum}}$ condition
 - $Z8_{\text{sum}} > Z_{\text{peak}} * Z8_{\text{Filt_Scale}}$ (At panel core)

5 PSoC™ 6 application notes

- $Z8_sum > Z_peak * Z8_Filt_Scale/2$ (At panel edge)
- $Z8_sum > Z_peak * Z8_Filt_Scale/4$ (At panel corner)

Where,

Z_peak = Maximum signal obtained

$Z9_sum$ = Total sum of 3x3 difference array

$Z8_sum = Z9_Sum - Z_peak$

$Z9_Filt_Scale = (0.8 * Z9_Sum) / \text{Finger threshold}$

$Z8_Filt_Scale = (0.8 * Z8_Sum) / \text{Finger threshold}$

These values ensure that the detected object is of the correct proportions.

- $Z8_sum$ condition is checked to see if the relative mass of the finger is large enough to be recognized as a finger. This is done to discard very high noise in a segment, when the neighbouring sensors have no signal detected
- $Z9_sum$ condition is checked to see if the absolute mass of the finger is large enough to be recognized as a finger. Similar to the $Z8$ condition, the $Z9$ condition may prevent noise-induced false touches

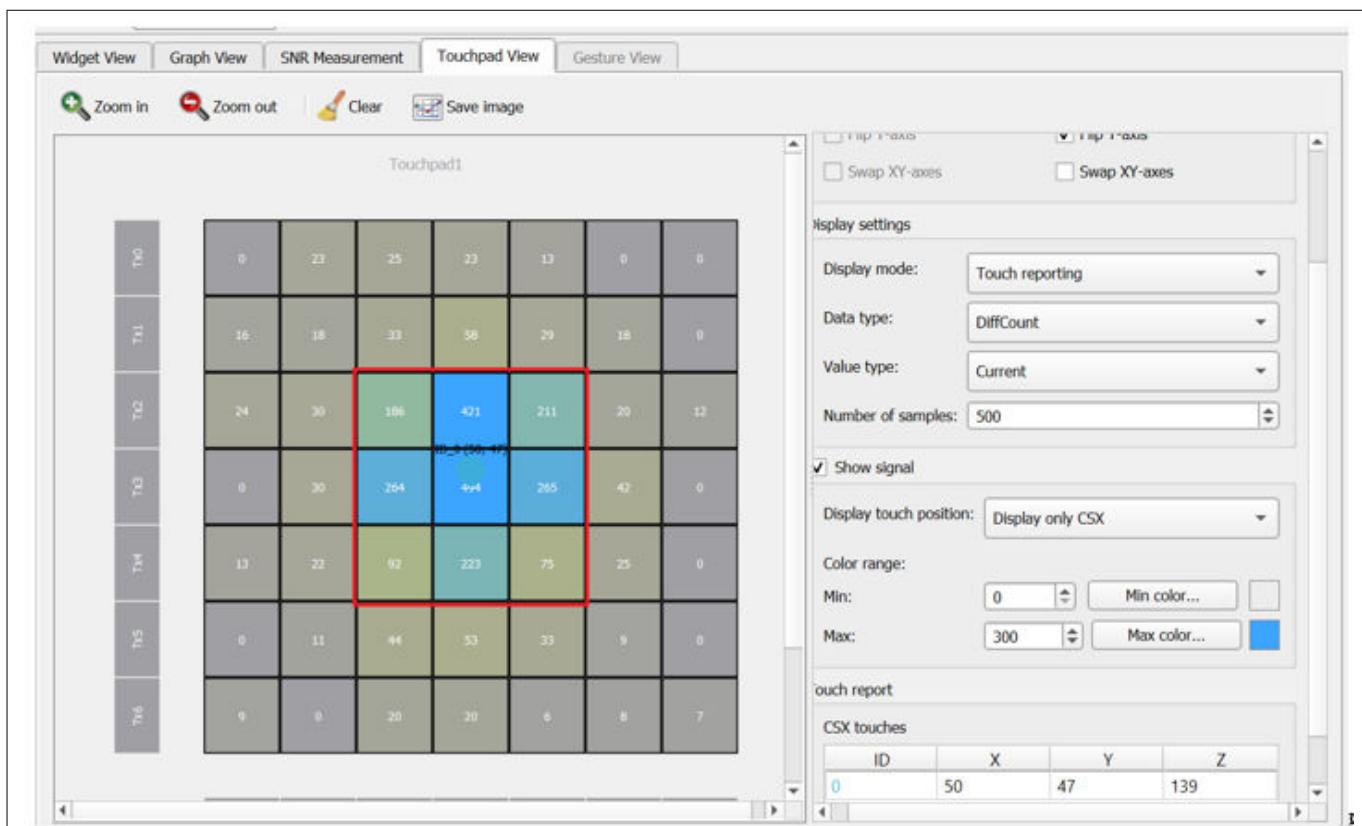


Figure 270 3x3 matrix obtained in CAPSENSE™ tuner

Figure 271 illustrates an overview of the CSX touchpad tuning procedure.

5 PSoC™ 6 application notes

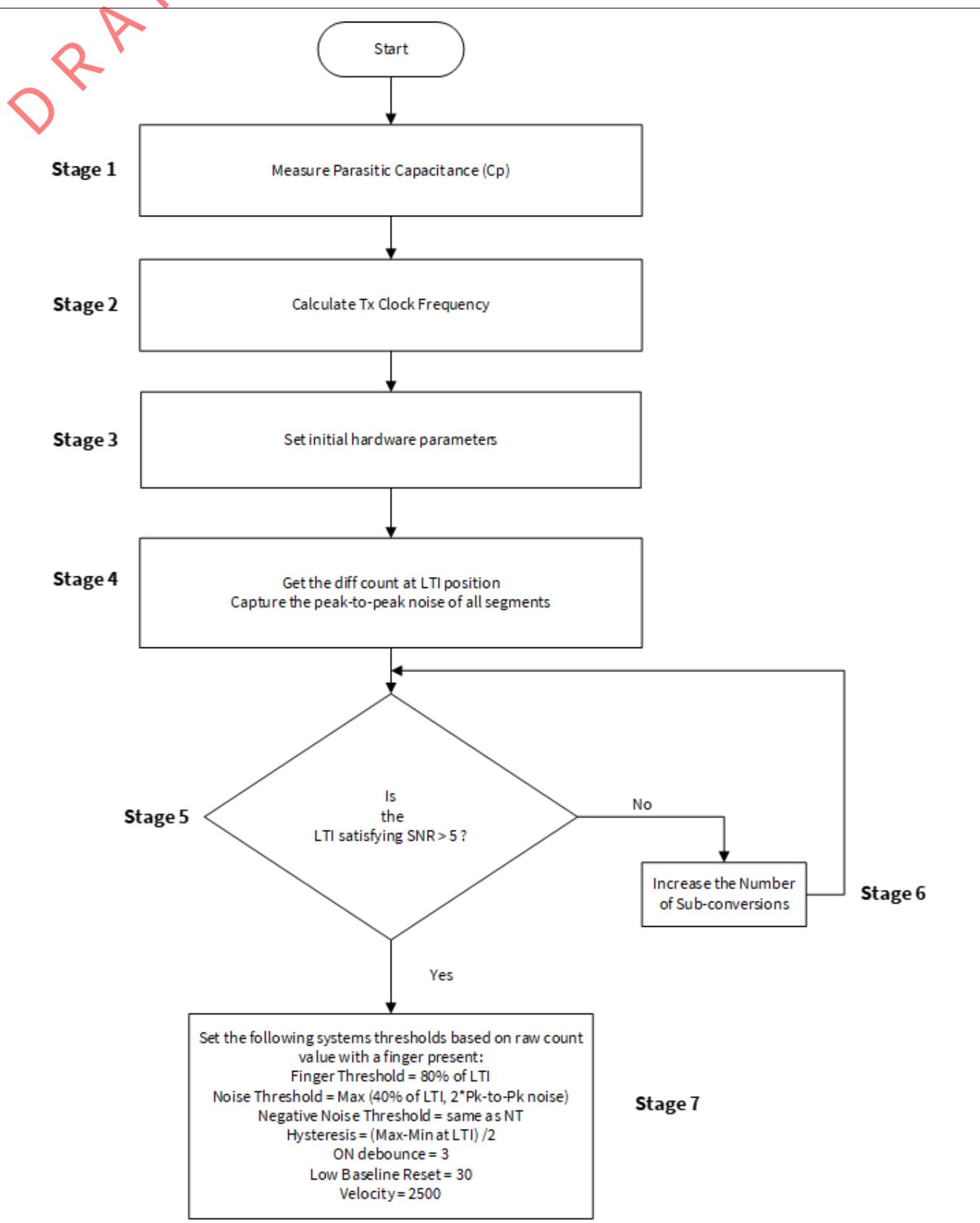


Figure 271 CSX touchpad widget tuning flowchart

LTI measures the peak diff-count when a finger touch is centered between the four sensors. The LTI signal count is the average of the four peak sensors. This gives the least valid touch signal.

~~5 PSoC™ 6 application notes~~

~~DRAFT~~ 5.8.5.3.4 CSD-RM sensing method (fifth-generation)

This chapter explains the basics of manual tuning using CSD-RM sensing method (Fifth-Generation). It also explains the hardware and software parameters that influence the CSD-RM sensing method and the procedure of manual tuning for button, slider, touchpad and proximity widgets.

Basics

Conversion gain and CAPSENSE™ signal

Conversion gain will influence how much signal the system sees for a finger touch on the sensor. If there is more gain, the signal is higher, and a higher signal means a higher achievable **Signal-to-noise ratio (SNR)**.

Note: *An increased gain may result in an increase in both signal and noise. However, if required, you can use firmware filters to decrease noise. For details on available firmware filters, see [Table 41](#).*

Conversion gain in single CDAC

In the single CDAC mode, the raw count is directly proportional to the sensor capacitance.

$$\text{raw count} = G_{CSD} C_S$$

Equation 48 Raw count relationship to sensor capacitance

Where,

C_S = Sensor capacitance

$C_S = C_P$ if there is no finger present on sensor

$C_S = (C_P + C_F)$ when there is a finger present on the sensor

G_{CSD} = Capacitance to digital conversion gain of CAPSENSE™ CSD. The approximate value of this conversion gain according to [Equation 19](#) and [Equation 48](#) is shown using [Equation 49](#).

$$G_{CSD} = \text{MaxCount} \frac{1}{SnsClk_{Div} \cdot C_{ref}}$$

Equation 49 Capacitance to digital converter gain

Where,

The equation for raw count in the single CDAC mode, according to [Equation 49](#) and [Equation 48](#) is shown in [Equation 50](#).

$$\text{raw count} = N_{Sub} \frac{C_S}{C_{ref}}$$

Equation 50 Single CDAC mode raw counts

Where,

N_{Sub} = Number of sub-conversions

$SnsClk_{Div}$ = sense clock divider

C_S = Sensor capacitance

C_{ref} = Reference capacitance

~~5 PSoC™ 6 application notes~~

$$C_{ref} = \text{RefCDACCode} \times C_{lsb}$$

RefCDACCode = Reference CDAC value

$$C_{lsb} = 8.86fF$$

The tunable parameters of the conversion gain are C_{ref} , $SnsClk_{Div}$, and N_{Sub} . [Figure 272](#) shows a plot of raw count versus sensor capacitance.

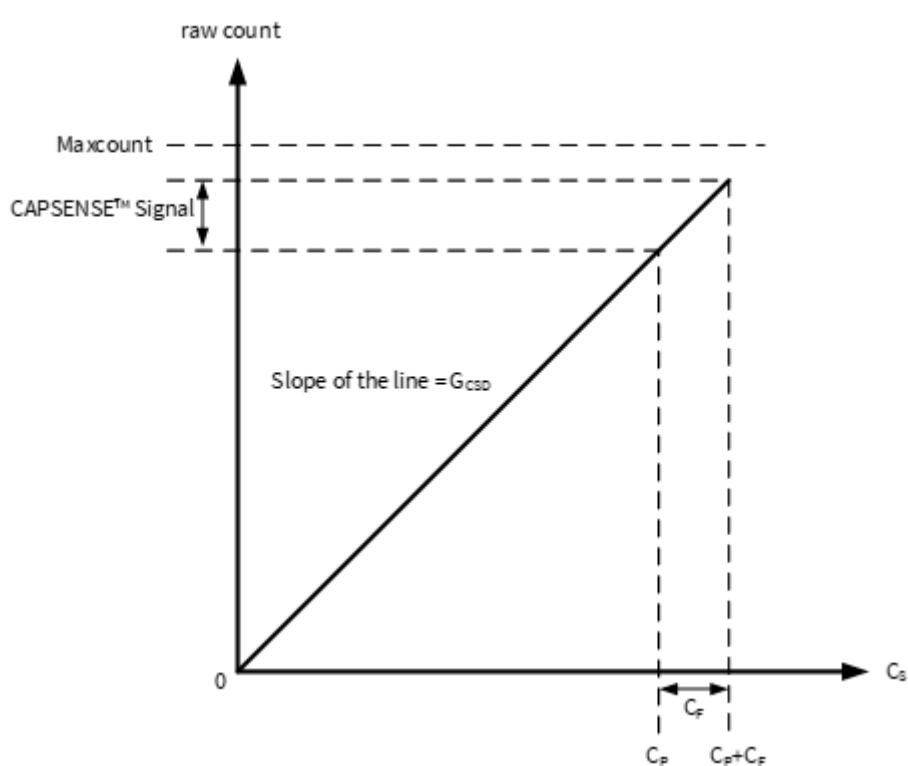


Figure 272 **Raw count versus sensor capacitance**

The change in raw counts when a finger is placed on the sensor is called CAPSENSE™ signal. [Figure 273](#) shows how the value of the signal changes with respect to the conversion gain.

5 PSoC™ 6 application notes

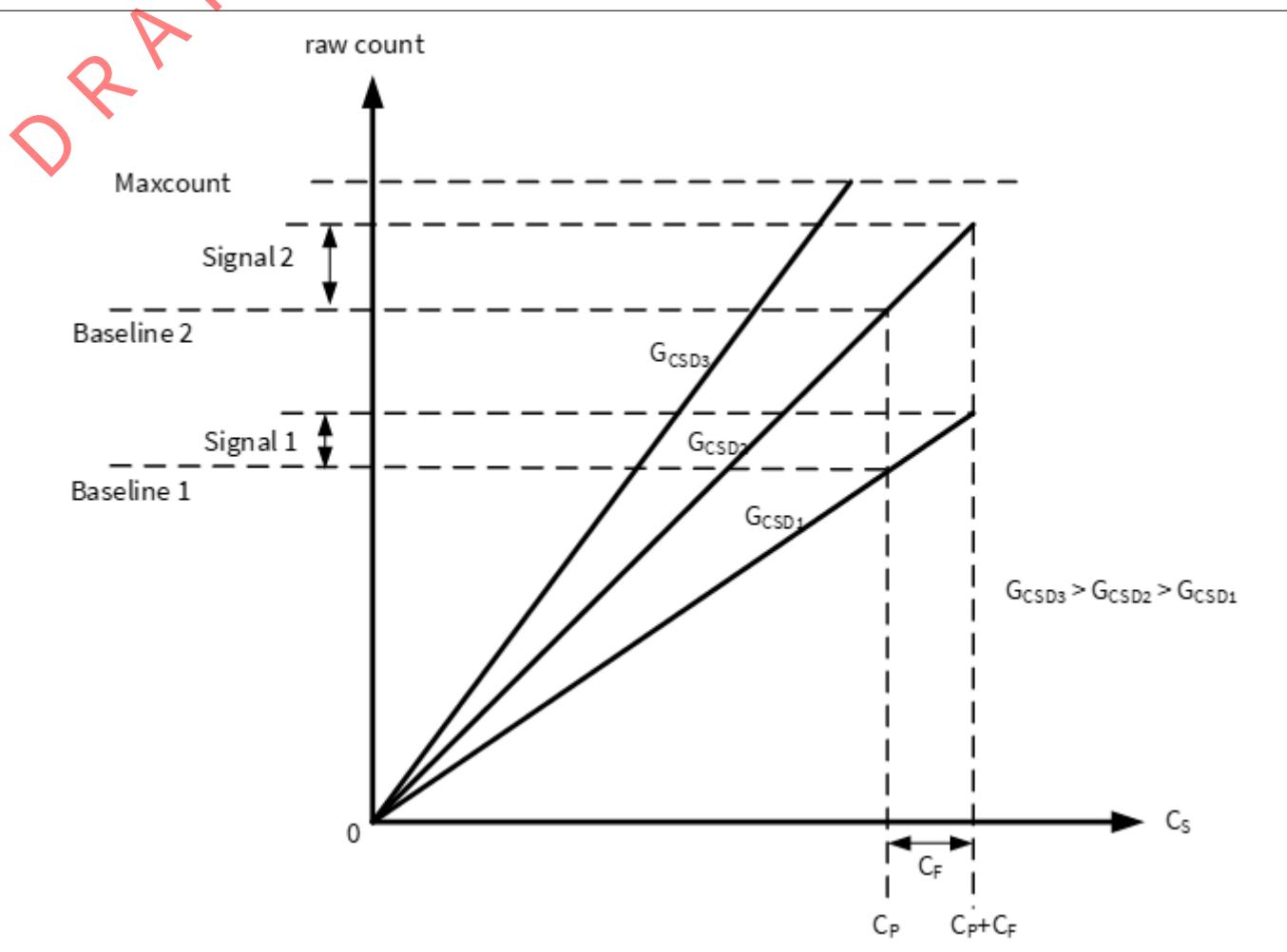


Figure 273 Signal values for different conversion gains

Figure 273 shows three plots corresponding to three conversion gain values G_{CSD3} , G_{CSD2} , and G_{CSD1} . An increase in the conversion gain results in higher signal value. However, this increase in the conversion gain also moves the raw count corresponding to C_P (that is, Baseline) towards the maximum value of raw count (Maxcount). For very high gain values, the raw count saturates as the plot of G_{CSD3} shows. Therefore, tune the conversion gain to get a good signal value while avoiding saturation of raw count. Tune the CSD-RM parameters such that when there is no finger on the sensor, that is when $C_S = C_P$, the raw count = 85% of Maxcount as Figure 274 shows. This ensures maximum gain, with enough margin for the raw count to grow because of environmental changes, and not saturate on finger touches.

5 PSoC™ 6 application notes

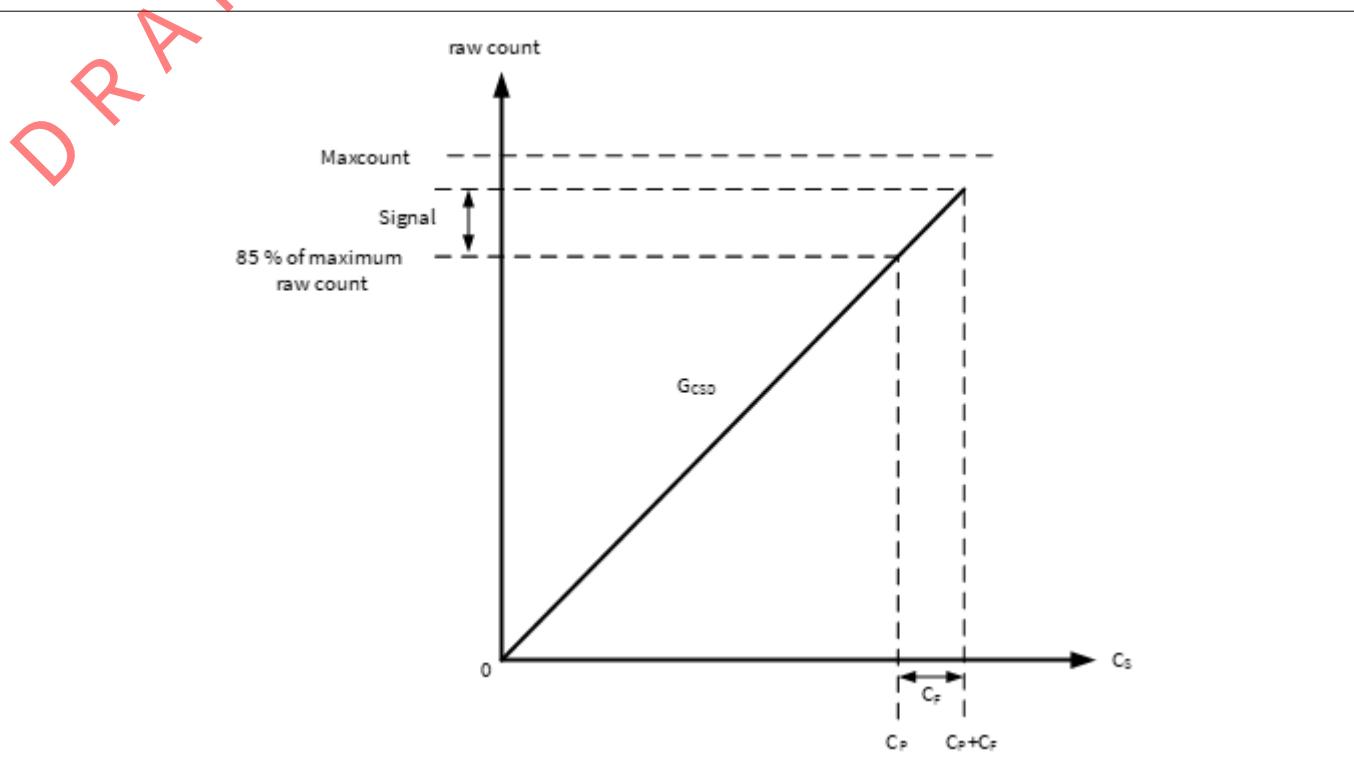


Figure 274 Recommended tuning

Conversion gain in dual CDAC mode

The equation for raw count in the dual CDAC mode, according to [Equation 20](#) and [Equation 48](#) is shown in [Equation 51](#).

$$\text{rawcount} = G_{\text{CSD}} C_S - \text{Maxcount} \times \frac{2 \times C_{\text{comp}}}{C_{\text{ref}} \text{CompCLK}_{\text{div}}}$$

Equation 51 Dual CDAC mode raw counts

Where,

$\text{Maxcount} = N_{\text{Sub}} * SnsClk_{\text{Div}}$

$SnsClk_{\text{Div}}$ = Sense clock divider

N_{Sub} = Number of sub-conversions

C_{ref} = Reference capacitance

C_{comp} = Compensation capacitance

$\text{CompCLK}_{\text{Div}}$ = CDAC compensation divider

C_S = Sensor capacitance

$C_{\text{ref}} = \text{RefCDACCCode} * C_{\text{lsb}}$

$C_{\text{comp}} = \text{CompCDACCCode} * C_{\text{lsb}}$

RefCDACCCode = Reference CDAC value

CompCDACCCode = Compensation CDAC value

$C_{\text{lsb}} = 8.86\text{fF}$

G_{CSD} is given by [Equation 49](#).

5 PSoC™ 6 application notes

In both single CDAC and dual CDAC mode, tune the CSD-RM parameters, so that when there is no finger on the sensor, that is when $C_S = C_P$, the raw count = 85% of Maxcount, as Figure 275 shows, to ensure high conversion gain, to avoid Flat-spots, and to avoid raw count saturation due to environmental changes.

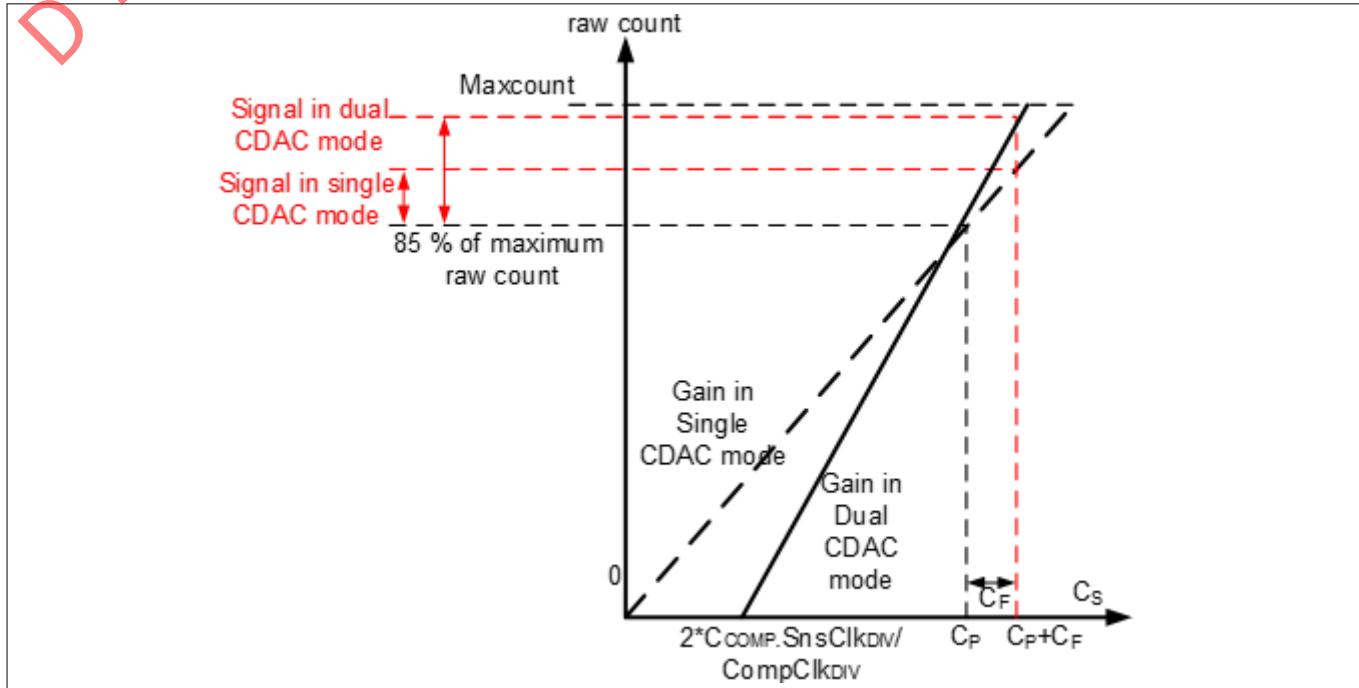


Figure 275 Recommended tuning in dual CDAC mode

As Figure 275 shows, the 85% requirement restricts to a fixed gain in single-CDAC mode, while in dual-CDAC mode, gain can be increased by moving the C_S axis intercept to the right (by increasing $CompClk_{DIV}$) and correspondingly decreasing the modulator CDAC switching ($SnsClk_{DIV}$) to still achieve raw count = 85% of Maxcount for $C_S = C_P$. Using dual CDAC mode this way brings the following changes to the Raw Count versus C_P graph:

- Use of compensation CDAC introduces a non-zero intercept on the C_S axis as shown in Equation 52

$$C_S \text{ axis intercept} = \left(\frac{2 \times C_{comp} SnsClk_{Div}}{CompClk_{Div}} \right)$$

Equation 52 C_S axis intercept with regards to C_{comp}

- The value of C_{ref} in the dual CDAC mode is half compared to the value of C_{ref} in the single CDAC mode (all other parameters remaining the same), so the gain G_{CSD} in the dual CDAC mode is double the gain in the single CDAC mode according to Equation 26. Thus, the signal in the dual CDAC mode is double the signal in the single CDAC mode for a given number of sub-conversions (N_{sub}).

While manually tuning a sensor, refer Equation 26, Equation 28 and the following points:

1. Higher gain leads to increased sensitivity and better overall system performance. However, do not set the gain such that raw counts saturate, as the plot of gain G_{CSD3} shows in Figure 249. It is recommended to set the gain in such a way that the raw count corresponding to C_P is 85 percent of the maximum raw count for both the single CDAC and dual CDAC mode
2. The sense clock frequency (F_{SW}) should be set carefully; higher the frequency, higher the C_S sampling rate which allows for more F_{SW} periods and better noise averaging, but the frequency needs to be low enough to fully charge and discharge the sensor as Equation 32 indicates
3. Enabling the compensation CDAC (baseline compensation) plays a huge role in increasing the gain; it will double the gain if set as recommended in Conversion gain in dual CDAC mode. Always enable Compensation CDAC, make sure the calibrated C_{ref} is in valid range when enabling Compensation CDAC

~~5 PSoC™ 6 application notes~~

- ~~4.~~ Lower the reference CDAC, higher the gain. Adjust your CDAC to achieve the highest gain, but make sure that the raw counts corresponding to C_p have enough margin for environmental changes such as temperature shifts, as indicated in [Figure 250](#) and [Figure 251](#)
- ~~5.~~ Increasing the number of sub-conversions used for scanning increases gain. An increase in number of sub-conversions also increases the scan time according to [Equation 9](#). A balance of scan time and gain need to be achieved using number of sub-conversions (N_{sub})

Flat-spots

Ideally, raw counts should have a linear relationship with sensor capacitance as [Figure 248](#) and [Figure 251](#) show. However, in practice, RM converter has non-sensitivity zones, also called flat-spots or dead-zones – for a range of sensor capacitance values, the RM converter may produce the same raw count value as [Figure 276](#) shows. This range is known as a dead-zone or a flat-spot.

[Equation 53](#) shows the flat-spots relation to different CAPSENSE™ hardware parameters.

$$\text{Flatspots Width} \propto \frac{C_s^2}{C_{MOD}} \cdot \frac{F_{SW}}{F_{MOD} \cdot \text{Bal\%}}$$

Equation 53 Flat-spots width

Where,

C_s = Sensor capacitance

C_{MOD} = Modulator capacitor

F_{SW} = Frequency of the sense clock

F_{MOD} = Modulator clock frequency

Bal% = Rawcount calibration percentage

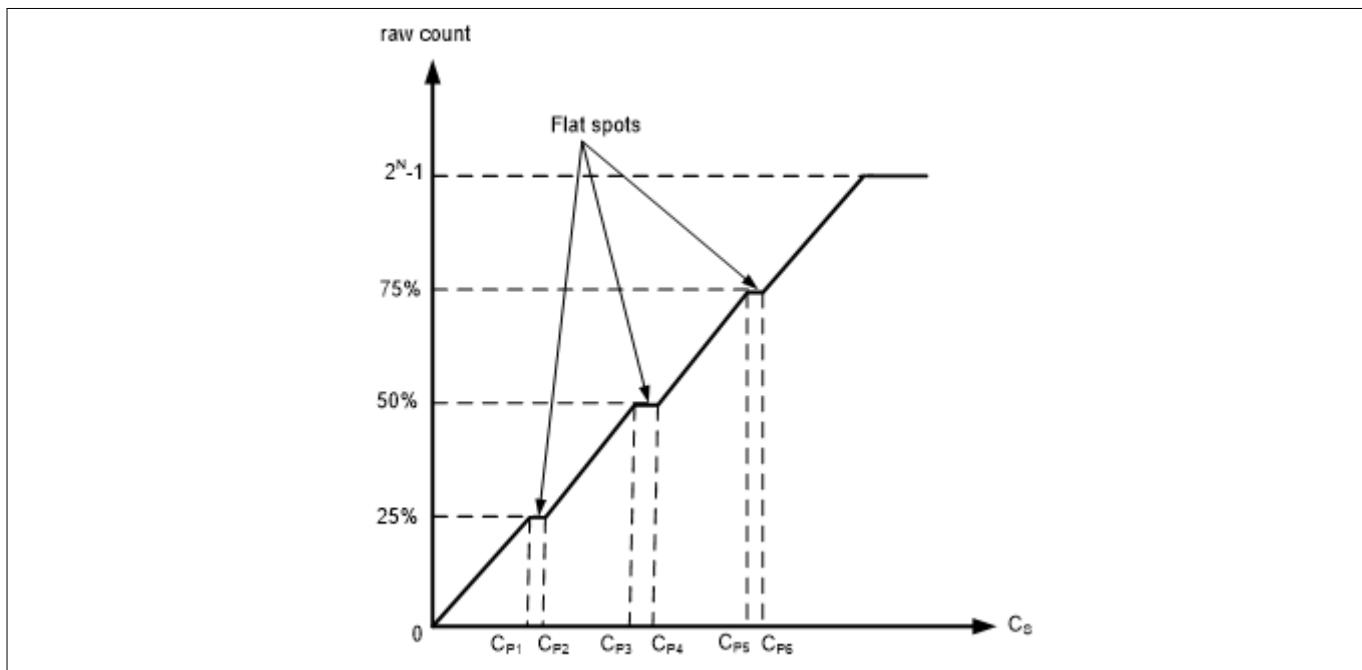


Figure 276 Flat-spots in raw counts versus sensor capacitance when direct clock is used

Flat-spots reduction techniques

1. Set rawcount calibration to 85%

~~5 PSoC™ 6 application notes~~

As per [Equation 53](#), flat-spots width is inversely proportional to calibration level. Setting calibration to 85% decrease the width of flat-spots significantly.

~~2. Enable dithering~~

An additional Dither CDAC is available in Fifth-Generation CAPSENSE™ architecture, which adds white noise that moves the conversion point around the flat-spots region.

~~3. Enable PRS clock~~

These flat-spots are prominent when direct clock is used as [Sense Clock](#) source. Flat-spots are reduced if PRS is used as the sense clock source (see also section [Using SmartSense to determine hardware parameters](#)). PRS clock can results in a slight reduction of signal or sensitivity at higher rawcount calibration. Recommended to set the rawcount calibration to 65% when PRS is used as clock source.

~~4. Use larger C_{MOD}~~

The flat-spots width is inversely proportional to the C_{MOD} used. Fifth-Generation architecture supports C_{MOD} upto 10 nF and typical value is 2.2 nF. And increasing C_{MOD} have the adverse effect of increasing the noise, initialization time and minimum signal required to detect.

~~5. Increase sense clock divider~~

Increasing sense clock divider decreases flat-spots width but increases the scan time. If the flat-spot is detected, increase the Sense clock divider such that the scan time requirement is met.

[Table 47](#) lists different the flat-spots reduction techniques in recommended priority and other considerations.

Table 47 Flat-spots reduction techniques

S. No	Flat-spots reduction techniques	Recommendation	Additional benefits	Disadvantage
1	Set rawcount calibration to 85%.	High	Improves sensitivity	-
2	Enable dithering	High	-	-
3	Enable PRS clock	Low	Improves EMI/EMC radiation and susceptibility	Needs to set rawcount calibration to 65%. Decreases sensitivity.
4	Increase C_{MOD}	Low	-	Increases noise
5	Increase sense clock divider	Low	-	Increases scan time

Selecting CAPSENSE™ hardware parameters

CAPSENSE™ hardware parameters govern the conversion gain and CAPSENSE™ signal. [Table 48](#) lists the CAPSENSE™ hardware parameters that apply to CSD-RM sensing method. The following subsections provide guidance on how to adjust these parameters to achieve optimal performance for CAPSENSE™ CSD-RM system.

Table 48 CAPSENSE™ component hardware parameters

S. No	CAPSENSE™ parameter in ModusToolBox™
1	Scan mode
2	Scan connection method
3	Number of Init sub-conversions
4	Sense clock divider

(table continues...)

5 PSoC™ 6 application notes

Table 48 (continued) CAPSENSE™ component hardware parameters

S. No	CAPSENSE™ parameter in ModusToolBox™
5	Sense clock source
6	Modulator clock divider
7	Reference CDAC value
8	CDAC compensation divider
9	Compensation CDAC value
10	Number of sub-conversions
11	Enable CDAC dither

[5.3.4.2.1](#) and [5.3.4.2.2](#) show selecting the CAPSENSE™ parameters in Eclipse IDE for ModusToolbox™. For more details on configuring CAPSENSE™, see the [Component datasheet/middleware document](#).

Using SmartSense to determine hardware parameters

Table 48 lists the CAPSENSE™ hardware parameters. Tuning these parameters manually for optimal value is a time-consuming task. You can use SmartSense to determine these hardware parameters and take it as an initial value for manual tuning. You can fine-tune these values to further optimize the scan time, SNR, power consumption, or improving EMI/EMC capability of the CAPSENSE™ system. Set the tuning mode to SmartSense and configure default values for parameters other than finger capacitance, [Sense clock source](#) and [CDAC dither](#). Set these as per the application requirement.

See the [SmartSense](#) section for the tuning procedure and use the Tuner GUI to read back all the hardware parameters set by SmartSense. See the [CAPSENSE™ tuner guide](#) for more details on how to use the Tuner GUI.

[Figure 277](#) shows the best hardware parameter values in the Tuner GUI that are tuned by SmartSense for a specific hardware to sense a minimum finger capacitance of 0.1 pF.

5 PSoC™ 6 application notes

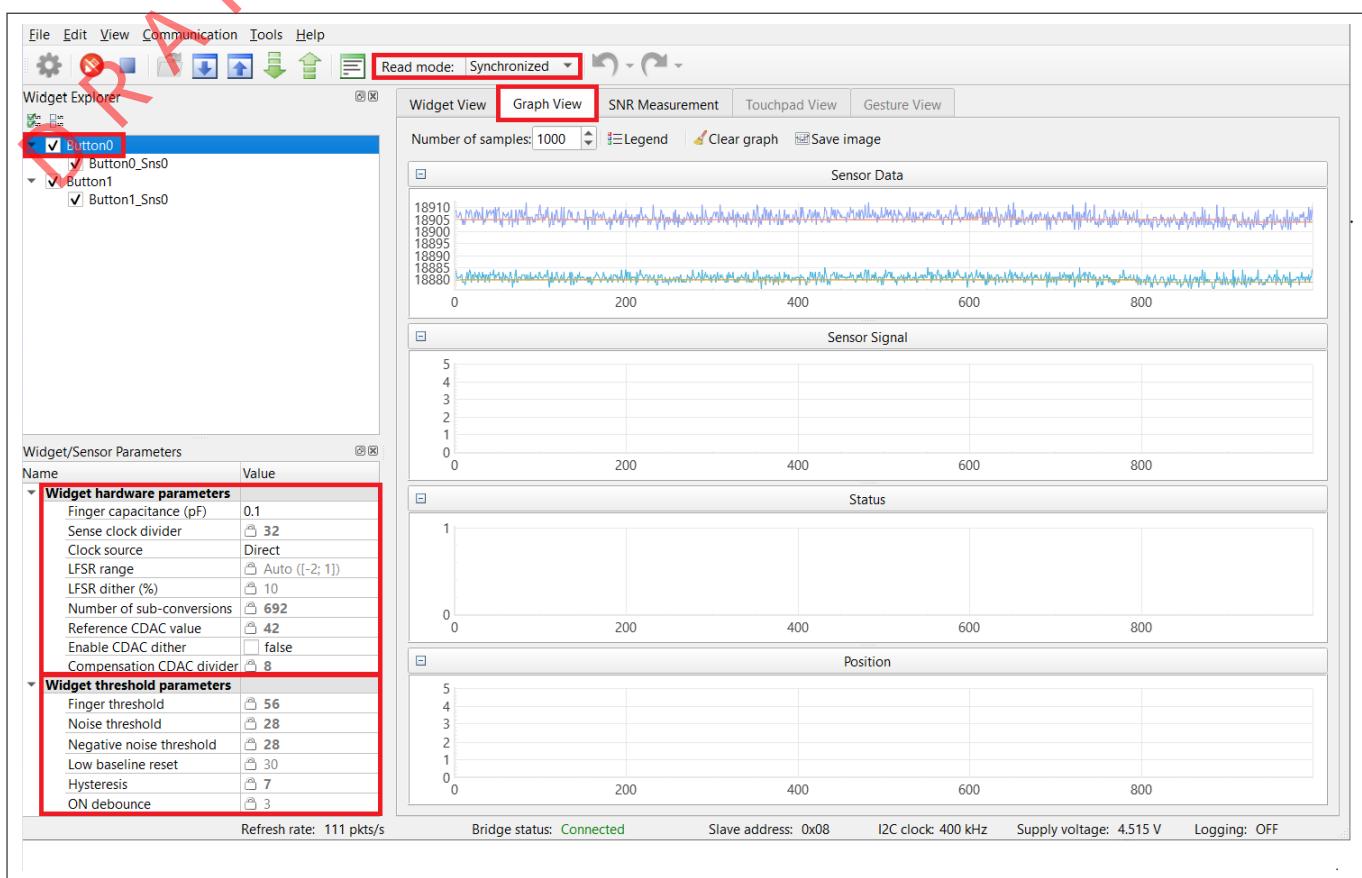


Figure 277 Read-back hardware parameter values in tuner GUI

Manually tuning hardware parameters

Scan mode

Scan mode can be set as CS-DMA or Interrupt Driven mode. For autonomous scanning select DMA mode and for legacy interrupt based scanning select Interrupt Driven mode.

Sensor connection method

Autonomous scanning is only available in CTRLMUX method, but the numbers of supported pins are limited in this method (see the [Device datasheet](#) for supported pins). Additionally provides better immunity to on-chip IO noise. Choose AMUXBUS method to support more number of pins in Interrupt Driven mode.

Modulator clock frequency

The modulator clock governs the conversion time for capacitance-to-digital conversion, also called the “sensor scan time” (see [Equation 9](#)).

A lower modulator clock frequency implies the following:

- Longer conversion time (see [Equation 55](#) and [Equation 58](#)).
- Lower peak-to-peak noise on raw count because of longer integration time of the ratiometric converter
- Wider [Flat-spots](#)

~~5 PSoC™ 6 application notes~~

Select the highest frequency for the shortest conversion time and narrower flat-spots for most cases. Use slower modulator clock to reduce peak-to-peak noise in raw counts if required.

Based on the required Modulator clock frequency (F_{MOD}), calculate the modulator clock divider using [Equation 54](#).

$$\text{Modulator clock divider} = \frac{F_{Clock}}{F_{MOD}}$$

Equation 54 Modulator clock divider

Where,

F_{Clock} = Clock frequency connected to CAPSENSE™ block

Initialization sub-conversions

As part of initialization, C_{MOD1} and C_{MOD2} needs to be charged at required voltage (VDDA/2). There are three phases in initialization – C_{MOD} initialization, C_{MOD} short and initialization sub-conversions. During C_{MOD} initialization phase C_{MOD1} is pulled to GND and C_{MOD2} is pulled to VDDA. During C_{MOD} short phase both capacitors are tied together so the charge is shared to produce a voltage close to VDDA/2 on both. After the 2 phases the scanning is started but rawcount is discarded for number of init sub-conversions.

Number of init sub-conversions should be selected based on [Equation 55](#).

$$\text{Number of init subconversions} = \text{ceiling} \left(\frac{C_{MOD} \times V_{OS}}{VDDA \times C_S \times (1 - \text{Base \%}) \times \left(\frac{1}{\text{Bal \%}} - 1 \right)} \right) + 1$$

or

$$\text{Number of init subconversions} = \text{ceiling} \left(\frac{C_{MOD} \times V_{OS}}{VDDA \times SnsClkDiv \times C_{ref} \times (1 - \text{Bal \%})} \right) + 1$$

Equation 55 Number of init sub-conversions

Where,

C_{MOD} = Modulator capacitor

V_{OS} = Comparator offset voltage (3 mV)

C_S = Sensor capacitance

Base% = Baseline compensation percentage

Bal% = Rawcount calibration percentage

$SnsClk_{Div}$ = sense clock divider

C_{ref} = Reference capacitance

C_{ref} = RefCDACCode * C_{lsb}

RefCDACCode = Reference CDAC value

C_{lsb} = 8.86 fF

Sense clock parameters

There are two parameters that are related to Sense clock: Sense clock source and Sense clock divider.

~~DEAF~~ 5 PSoC™ 6 application notes

Sense clock source

Select “Auto” to let the Component automatically choose the best Sense clock source from Direct, PRSx, and SSCx for each widget. If not selecting Auto, select the clock source based on the following:

- Use SSCx (spread spectrum clock) modes for reducing EMI/EMC noise at a particular frequency. This feature is available in PSoC™ 4S-Series, PSoC™ 4100S Plus, PSoC™ 4100PS, PSoC™ 4100S Max and PSoC™ 6 family of devices. In this case, the frequency of the sense clock is spread over a predetermined range
- Use Direct clock for absolute capacitance measurement
- Use PRSx (pseudo random sequence) modes to remove flat-spots and improve EMI/EMC radiation and susceptibility. In 5th Generation CAPSENSE™, PRS clock results in a slight reduction in signal/sensitivity at higher rawcount calibration percent, hence 65% rawcount calibration is recommended when PRS clock is used

When selecting SSCx, you need to select the Sense clock frequency, Modulator clock frequency, and number of sub-conversion such that the conditions mentioned in [ModusToolbox™ CAPSENSE™ configurator guide](#) for SSCx clock source selection are satisfied.

Sense clock divider

The sense clock divider should be selected so that the sensor will charge and discharge completely in each sense clock period as [Figure 236](#) shows.

Note: For Fifth-Generation CSD-RM charging and discharging happens twice in a single clock period.

This requires that the maximum sense clock frequency be chosen per [Equation 56](#).

$$F_S(\text{maximum}) = \frac{1}{4 \times 5 \times R_{\text{SeriesTotal}} C_P}$$

Equation 56 Sense clock maximum frequency

$$R_{\text{SeriesTotal}} = R_{\text{EXT}} + R_{\text{internal}}$$

Equation 57 Total series resistance

Where,

C_p = Sensor parasitic capacitance

$R_{\text{SeriesTotal}}$ = Total series-resistance, including the R_{internal} resistance of the internal switches, the recommended external series resistance of 560 Ω (connected on PCB trace connecting sensor pad to the device pin), and trace resistance if using highly resistive materials (example ITO or conductive ink).

R_{internal} = Internal resistance, this varies based on scan and shield modes, see table [Table 49](#).

Table 49 Internal resistance for sensor

Scan mode	R_{internal}
CTRLMUX	525 Ω
AMUXBUS	425 Ω

The value for C_p can be estimated using the CSD Built-in-Self-test APIs. See the [Component datasheet/middleware document](#) for details.

To minimize the scan time, as [Equation 58](#) shows, it is recommended to use the maximum sense clock frequency (F_{SW}) satisfies the criteria as per [Equation 56](#).

5 PSoC™ 6 application notes

$$T_{SCAN} = \frac{N_{Sub}}{F_{SW}}$$

~~DRAFT~~ Equation 58 Sensor scan time

Where,

N_{Sub} = Number of sub-conversions

Based on required sense clock frequency (F_{SW}), the sense clock divider be chosen per [Equation 59](#).

$$\text{Sense clock divider} = \frac{F_{MOD}}{F_{SW}}$$

~~DRAFT~~ Equation 59 Sense clock divider

[Equation 49](#) shows that it is best to use the maximum clock frequency to have a good gain ; however, you should ensure that the sensor capacitor fully charges and discharges as shown in [Figure 236](#). And keep in mind that higher clock frequency increases current consumption as there are more charging and discharging.

Generally, the C_P of the shield electrode will be higher compared to sensor C_P . For good liquid tolerance, the shield signal should satisfy the condition mentioned in [Tuning shield electrode](#) chapter . If it is not satisfied, reduce the sense clock frequency further to satisfy the condition.

Number of sub-conversions

The number of sub-conversions decides the sensitivity of the sensor and sensor scan time. From [Equation 20](#) for a fixed modulator clock and Sense clock, increasing the number of sub-conversions (N_{Sub}) increases the signal and SNR. However, increasing the number of sub-conversions also increases the scan time of the sensor per [Equation 60](#).

$$\text{Scan time} = \frac{N_{Sub}}{F_{SW}}$$

~~DRAFT~~ Equation 60 CSD-RM scan time

Initially, set the value to a low number, and use the Tuner GUI to find the SNR of the sensor. If the SNR is not > 5:1 with the selected , try to increase the in steps such that the SNR requirement is met.

Capacitive DACs

Fifth-Generation CAPSENSE™ supports two CDACs: Reference CDAC (C_{ref}) and Compensation CDAC (C_{comp}) that balance C_{MOD} 's as [Figure 234](#) shows. These govern the [Conversion gain and CAPSENSE™ signal](#) Conversion gain and CAPSENSE™ for capacitance-to-digital conversion. The CAPSENSE™ Component allows the following configurations of the CDACs:

- Enabling or disabling of Compensation CDACs
- Enabling or disabling of Auto-calibration for the CDACs
- Compensation CDAC divider, DAC code selection for Reference and Compensation CDACs if auto-calibration is disabled

Reference CDAC (Cref)

The reference CDAC is used to compensate the charge transferred by the sensor self-capacitance (C_S) from C_{MOD} . The number of times it is switched depends on the self-capacitance of sensor. In case of finger placed over the sensor, additional reference CDAC switching is required to compensate.

~~5 PSoC™ 6 application notes~~

~~DRAFT~~ C_{ref} should satisfy below critieria:

- For Compensation Disabled:
 $RefCDACCode \geq 25$
- For Compensation Enabled:

$$RefCDACCode \geq \frac{20}{CDAC \text{ Compensation Divider}}$$

Where,

C_{ref} = Reference capacitance = $RefCDACCode * C_{lsb}$

$RefCDACCode$ = Reference CDAC value

$C_{lsb} = 8.86 \text{ fF}$

Compensation CDAC (Ccomp)

Enabling the compensation CDAC is called “dual CDAC” mode, and results in increased signal as explained in [Conversion gain and CAPSENSE™ signal](#). Enable the compensation CDAC for most cases.

The compensation capacitor is used to compensate excess self-capacitance from the sensor to increase the sensitivity. The number of times it is switched depends on the amount of charge the user application is trying to compensate (remove) from the sensor self-capacitance.

Reference CDAC (Cref)

The reference CDAC is used to compensate the charge transferred by the sensor self-capacitance (C_S) from C_{MOD} . The number of times it is switched depends on the self-capacitance of sensor. In case of finger placed over the sensor, additional reference CDAC switching is required to compensate.

C_{ref} should satisfy below critieria:

- For Compensation Disabled:
 $RefCDACCode \geq 25$
- For Compensation Enabled:

$$RefCDACCode \geq \frac{20}{CDAC \text{ Compensation Divider}}$$

Where,

C_{ref} = Reference capacitance = $RefCDACCode * C_{lsb}$

$RefCDACCode$ = Reference CDAC value

$C_{lsb} = 8.86 \text{ fF}$

Compensation CDAC (Ccomp)

Enabling the compensation CDAC is called “dual CDAC” mode, and results in increased signal as explained in [Conversion gain and CAPSENSE™ signal](#). Enable the compensation CDAC for most cases.

The compensation capacitor is used to compensate excess self-capacitance from the sensor to increase the sensitivity. The number of times it is switched depends on the amount of charge the user application is trying to compensate (remove) from the sensor self-capacitance.

Compensation CDAC divider

The number of times the compensation capacitor is switched in a single sense clock is denoted by K_{comp} . Select CDAC compensation divider based on [Equation 60](#) such that below criteria is satisfied.

1. CDAC compensation divider ≥ 4
2. K_{comp} should be a whole number

5 PSoC™ 6 application notes

DRAFT

$$\text{CDAC compensation divider} = \frac{\text{Sense clock divider}}{K_{comp}}$$

Equation 61

CDAC compensation divider

Auto-calibration

The auto-calibration feature enables the firmware to automatically calibrate the CDAC to achieve the required calibration target of 85%. It is recommended to enable auto-calibration for most cases. Enabling this feature will result in the following:

- Fixed raw count calibration to 85% of maximum raw count even with part-to-part C_p variation
- Decrease the effect of [Flat-spots](#)

If your design environment includes large temperature variation, you may find that the 85% CDAC calibration level is too high, and that the raw counts saturate easily over large changes in temperature, leading to lower SNR. In this case, adjust the calibration level lower by using `Cy_CapSense_CalibrateAllSlots()` in your firmware.

For proper functioning of CAPSENSE™ under diverse environmental conditions, it is recommended to avoid very low or high CDAC codes. For an 8-bit CDAC, it is recommended to use CDAC codes between 6-200 from the possible 0 to 255 range. You can use CAPSENSE™ tuner to confirm that the auto-calibrated CDAC values fall in this recommended range. If the CDAC values are out of the recommended range, based on [Equation 48](#), [Equation 49](#), and [Equation 51](#), you may change the Calibration level or F_{mod} or F_{SW} to get the CDAC code in proper range.

Disable CDAC auto-calibration if a change in C_p needs to be detected by measuring the raw count level at reset. For example:

- Detecting large variations in sensor C_p across boards or due to layout problems
- Detecting finger touch at reset
- Advanced CAPSENSE™ methods like liquid-level sensing, for example, to have different raw count level for different liquid levels at reset

Selecting CDAC codes

This is not the recommended approach. However, this approach could be used only if you want to disable auto-calibration for any reason. To get the CDAC code, you may first configure CAPSENSE™ Component with auto-calibration enabled and all other hardware parameters the same as required for final tuning and read back the calibrated CDAC values using [Tuner GUI](#). Then, re-configure the CAPSENSE™ Component to disable auto-calibration and use the obtained CDAC codes as fixed DAC codes read-back from the [Tuner GUI](#).

CDAC dither

As the input capacitance is swept, the raw count should increase linearly with capacitance. There are regions where the raw count does not change linearly with input capacitance these are called flat-spots, see section [Flat-spots](#) for more details. Dithering helps to reduce flat-spots using a dither CDAC. The dither CDAC adds white noise that moves the conversion point around the flat region.

Tuning shield electrode

The shield related parameters need to be additionally configured or tuned differently when you enable the Shield electrode in the CSD-RM sensing method for liquid tolerance or reducing the C_p of the sensor.

~~5 PSoC™ 6 application notes~~

~~Shield electrode tuning theory~~

Ideally, the shield waveform should be exactly the same as that of the sensor as explained in [CAPSENSE™ CSDRM shielding](#). However, in practical applications, the shield waveform may have a higher settling time. Observe the sensor and shield waveform in the oscilloscope; an example waveform is shown in [Figure 278](#) and [Figure 279](#). The shield waveform should settle to the sensor voltage within 90% of ON time of the sense clock waveform and the overshoot error of the shield signal with respect to VREF should be less than 10%.

If these conditions are not satisfied, you will observe a change in raw count of the sensors when touching the shield hatch; in addition, if inactive sensors are connected to shield as mentioned in [Inactive sensor connection](#), touching one sensor can cause change in raw count on other sensors, which indicates that there is cross talk if the shield electrode is not tuned properly.

Approximate maximum shield frequency (F_{Shield}) which ensures correct charging and discharging of shield waveform can be calculated using [Equation 62](#).

$$F_{\text{Shield}}(\text{maximum}) = \frac{1}{4 \times 5 \times R_{\text{SeriesTotal}} C_{\text{sh}}}$$

Equation 62 Sense clock maximum frequency

Where,

C_{sh} = Shield C_p

$R_{\text{SeriesTotal}} = R_{\text{internal}} + R_{\text{EXT}}$

R_{EXT} = External series resistor connected to shield electrode. Recommended value is 560 Ω .

R_{internal} = Internal resistance, this varies based on scan and shield modes, see [Table 50](#).

Table 50 Internal resistance for shield sensor

Scan Mode	$R_{\text{internal}} (\text{Active Shield})$	$R_{\text{internal}} (\text{Passive Shield})$
CTRLMUX	250 Ω	250 Ω
AMUXBUS	300 Ω	100 Ω

In SmartSense, the sense clock frequency is automatically set. Check if these conditions are satisfied. If not satisfied, switch to [Manual tuning](#) and set the sense clock frequency manually so that these conditions are satisfied.

5 PSoC™ 6 application notes

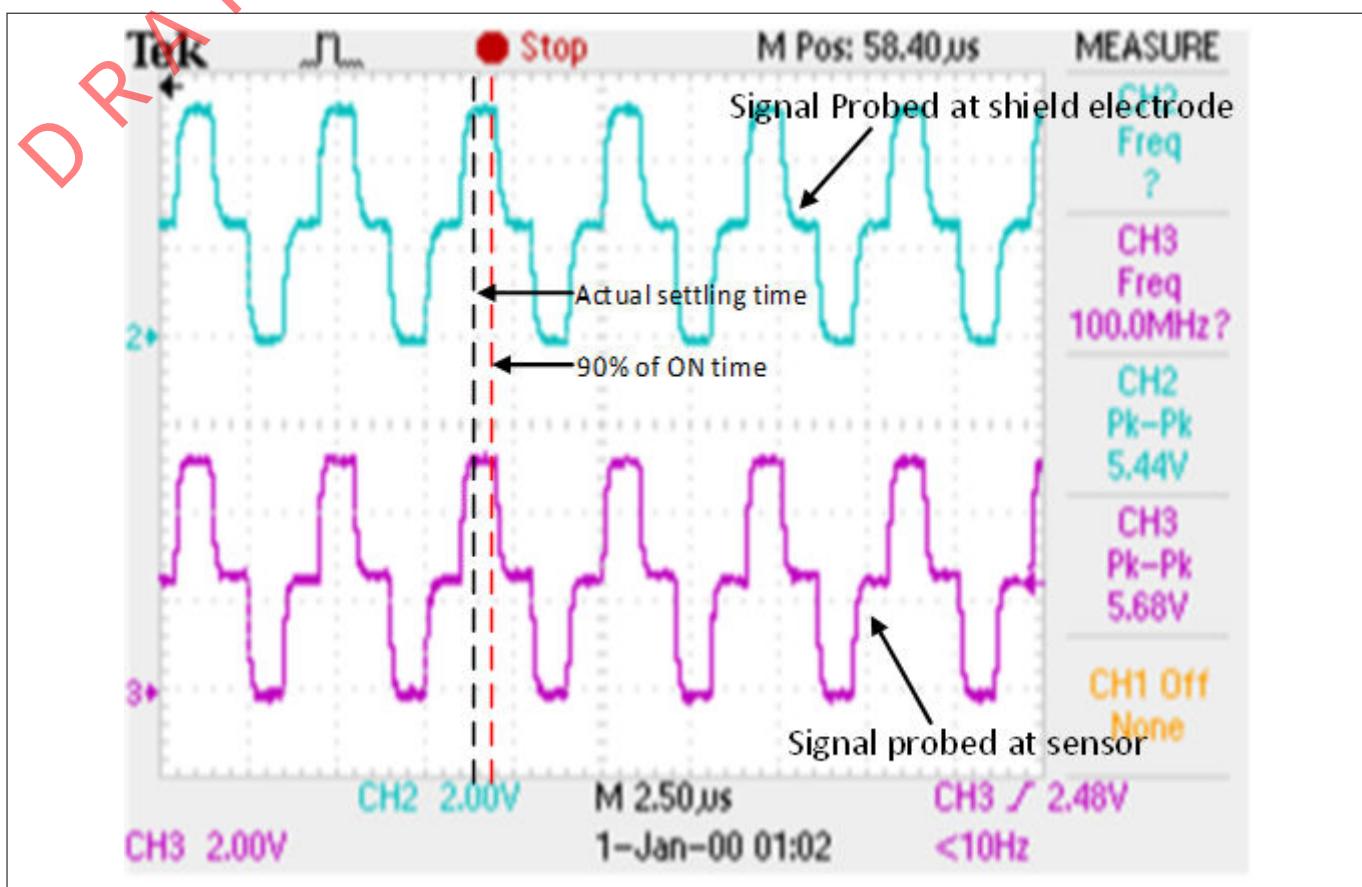


Figure 278 Properly tuned shield waveform (active shielding)

5 PSoC™ 6 application notes

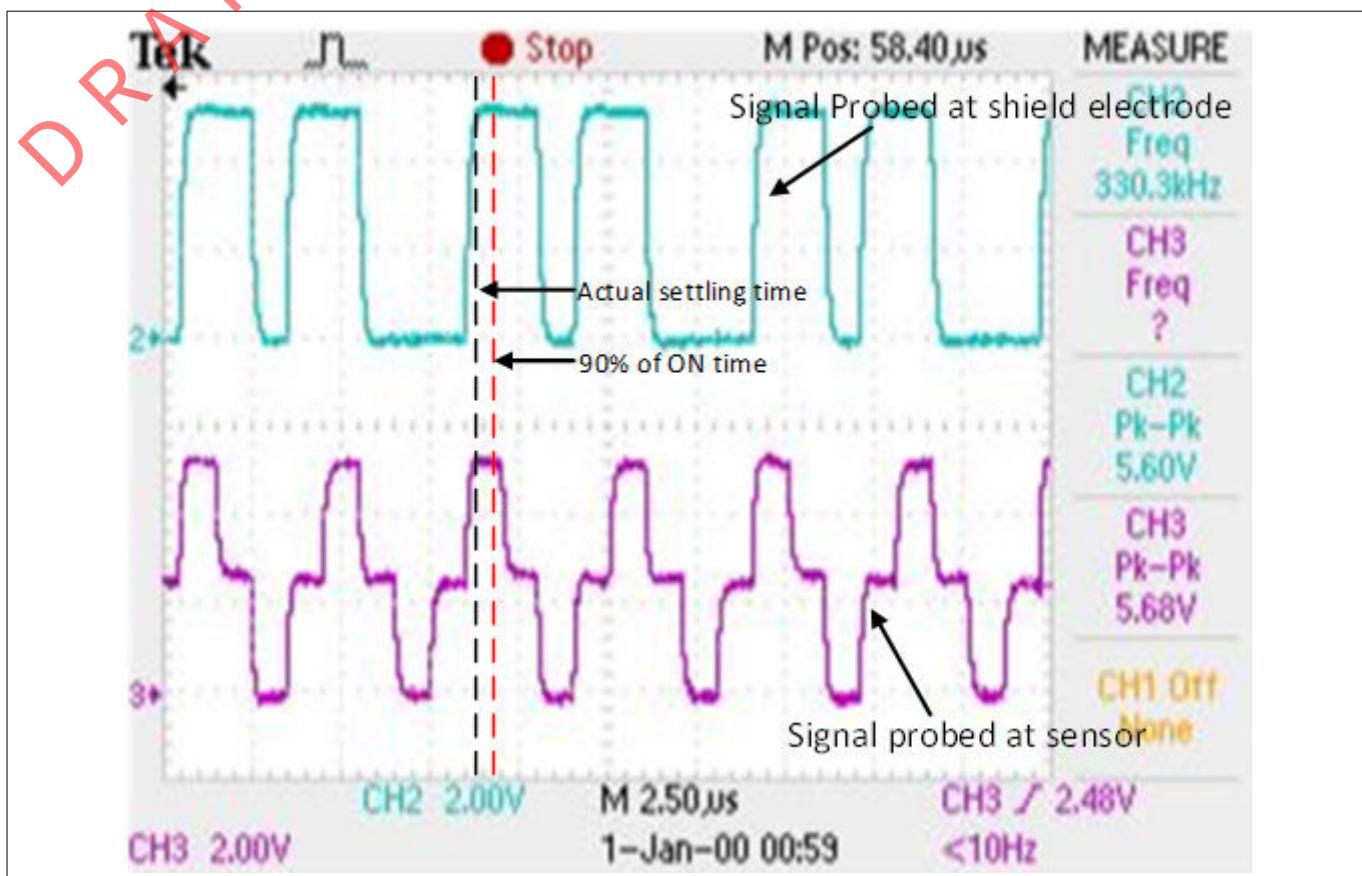


Figure 279 Properly tuned shield waveform (passive shielding)

Tuning shield-related parameters

Inactive sensor connection

When the shield electrode is enabled for liquid-tolerant designs, or if you want to use shield to reduce the sensor parasitic capacitance, this option should be specified as “Shield”; otherwise, select “Ground”.

However, there is a risk of higher radiated emission due to inactive sensors getting connected to Shield. In such situations, use the CAPSENSE™ API to manually control inactive sensor connections. Instead of connecting all unused sensors to the shield, connect only the opposing inactive sensors or inactive sensors closer to the sensor being scanned to shield for reducing the radiated emission.

Number of shield electrodes (total shield count)

This parameter specifies the number of shield electrodes required in the design. Most designs work with one dedicated shield electrode; however, some designs require multiple dedicated shield electrodes for ease of PCB layout routing or to minimize the PCB real estate used for the shield layer. See [Layout guidelines for shield electrode](#) Layout guidelines for shield .

Shield mode

The Fifth-Generation CAPSENSE™ architecture supports two shield modes – active and passive shielding. See [CAPSENSE™ CSDRM shielding](#) section to decide which mode is best suited for your application.

5 PSoC™ 6 application notes~~DEAF~~
Selecting CAPSENSE™ software parameters

CAPSENSE™ software parameters in Fifth-Generation are the same as that for Fourth-Generation; therefore, these parameters could be selected as mentioned in [Selecting CAPSENSE™ software parameters](#) section.

~~DEAF~~
Configuring autonomous scan

Autonomous scanning improves CPU offloading by removing the requirement of CPU intervention in between sensor scans. [Figure 280](#) shows the waveform of scanning all slots, which shows the CAPSENSE™ CPU bandwidth requirement for autonomous scanning and interrupt driven scanning. In autonomous scan once the CPU initiates a scan all slot command, there is no CPU interrupt is raised by CAPSENSE™ until all the slot scan is completed. But in interrupt driven scan, after each slot scan, a CPU interrupt is raised to configure the next slot sensors.

5 PSoC™ 6 application notes

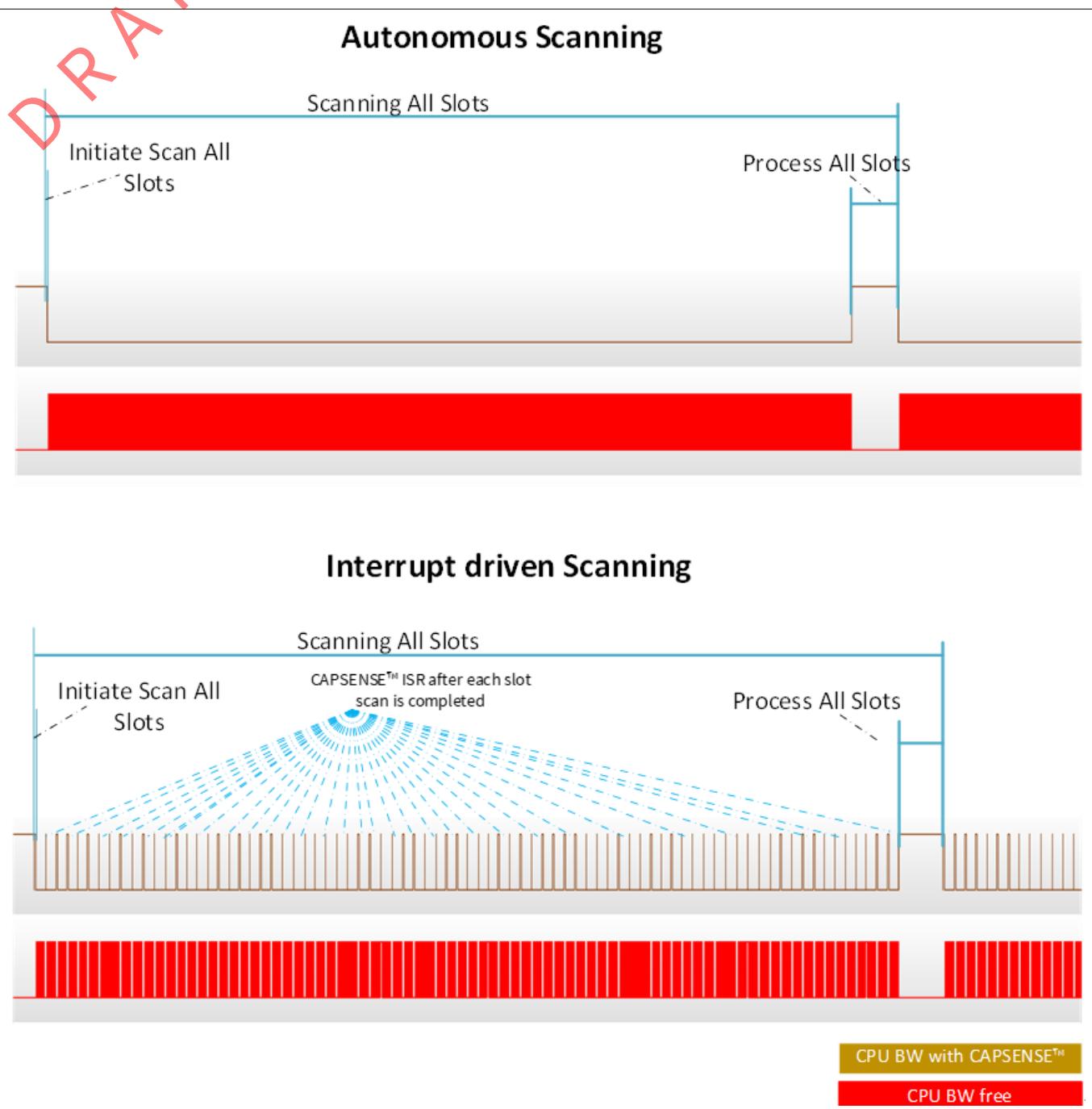


Figure 280 CAPSENSE™ configurator settings for scan mode

Autonomous scanning is only available when Scan mode is set as Chained Scanning – DMA (CS-DMA) in the CAPSENSE™ configurator as shown in [Figure 281](#). And sensor connection method is only available as CTRLMUX. This limits the number of available CAPSENSE™ sensors. In Interrupt driven mode, sensor connection can be configured as either AMUXBUS or CTRLMUX. Through AMUXBUS any GPIO pin can be configured as a CAPSENSE™ sensor, but CPU interrupts need to be serviced to configure every next sensor and read the scan result.

5 PSoC™ 6 application notes

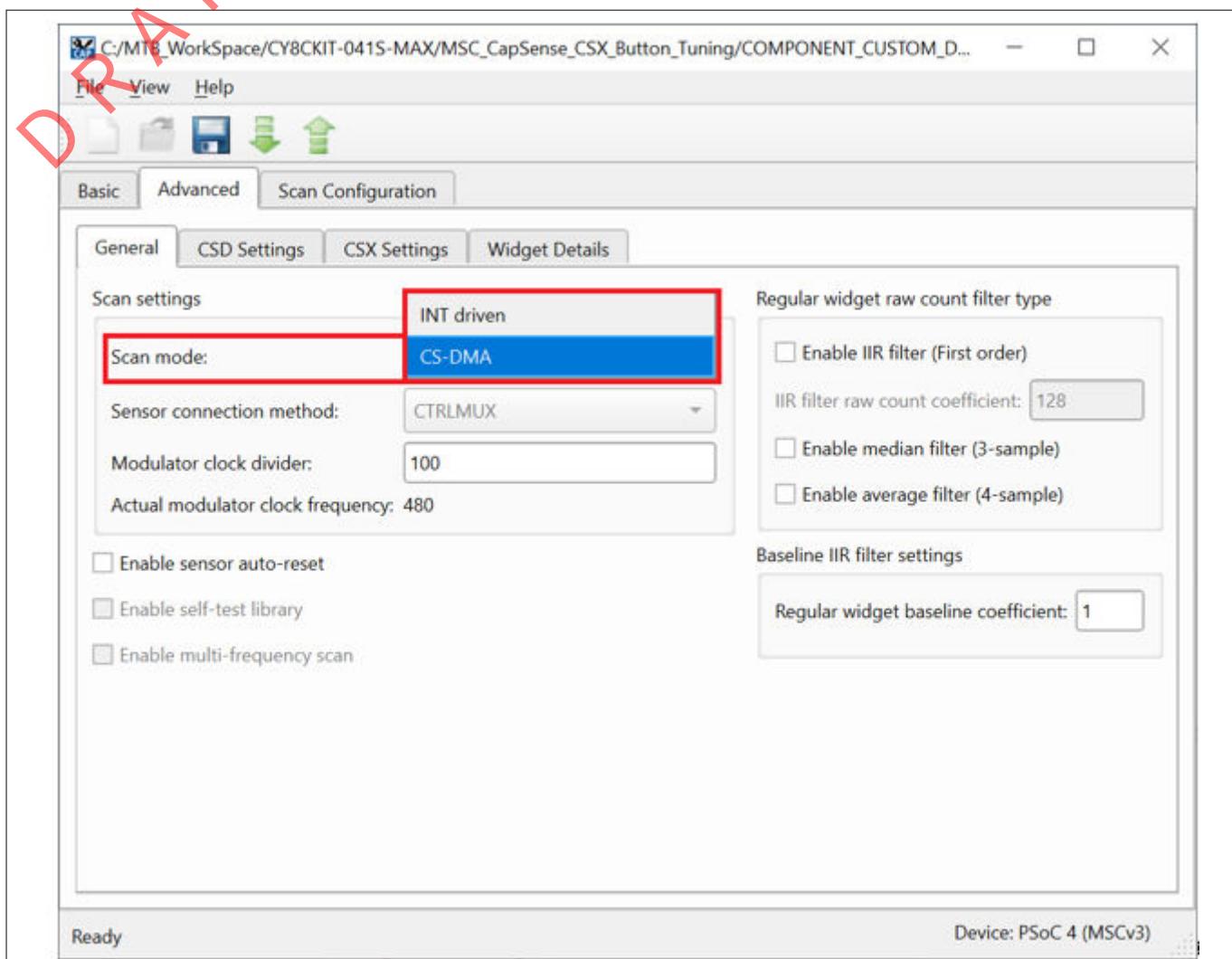


Figure 281 CAPSENSE™ configurator settings for scan mode

Chained scanning – DMA

In the chained scanning - DMA mode, DMA handles the configuration of each sensor, thereby avoiding the requirement of CPU intervention after each sensor scan completion. Each channel of MSC block requires four channels of DMA to be configured in the device configurator as shown in [Figure 282](#).

5 PSoC™ 6 application notes

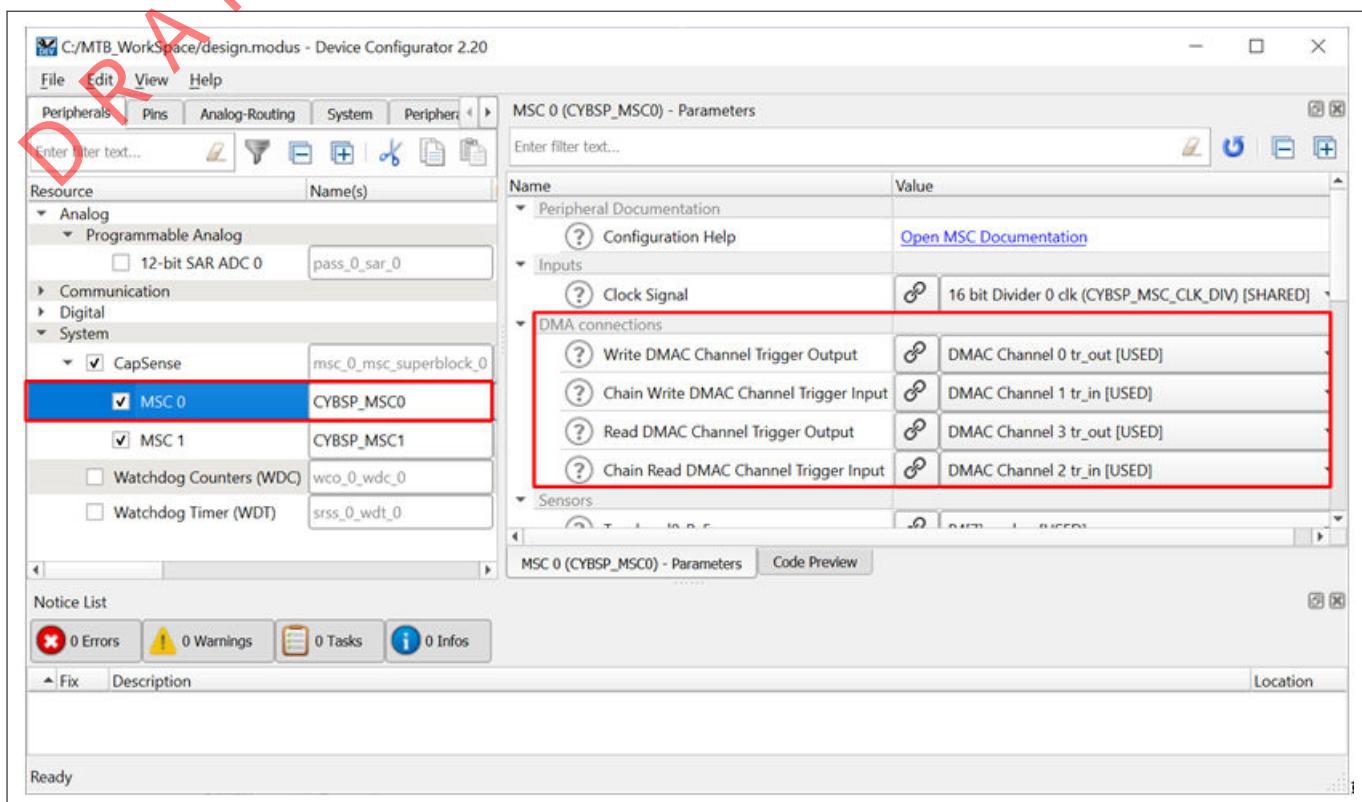


Figure 282 DMA configuration for MSC channelsCS-DMA scanning flow

Figure 283 illustrates the flow of CS-DMA based scanning mode.

1. Write DMA channel

Write DMA is configured to transfer scan configuration of a sensor to MSC block. Source address to the corresponding sensor's scan configuration is received from Chain Write DMA channel.

2. Chain Write DMA channel

When the MSC block completes scanning of current sensor, it will trigger the DMA to transfer the source address of next sensor's or first sensor's (if it is a new scan) scan configuration to the Write DMA channel.

3. Read DMA channel

Read DMA channel transfer the scan result (rawcount) to destination location of corresponding sensor.

4. Chain Read DMA channel

Once the current sensor scan is completed by MSC block, Chain Read DMA is triggered to transfer the destination location (address) of current sensor scan result (rawcount) to the Read DMA channel.

5 PSoC™ 6 application notes

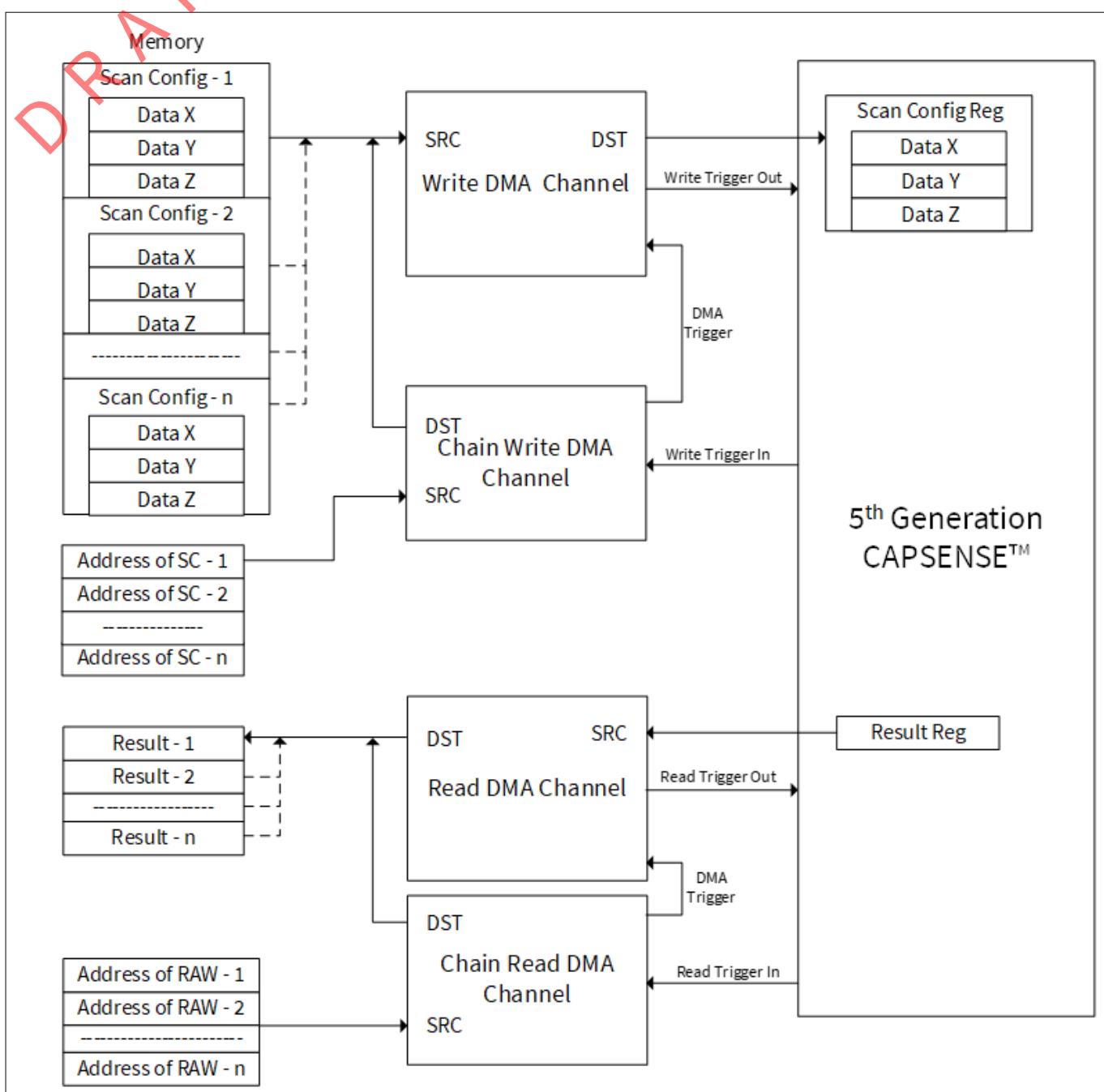


Figure 283 CS-DMA scanning flow

Multi-channel scanning

Multi-channel design uses both the instances of CAPSENSE™ MSC0 and MSC1, leading to simultaneous operation and reduction in scan time. Multi-channel scanning is in lock step thereby avoiding any cross-channel noise coupling. Scan synchronization is required to have the scanning in lock step. Fifth-Generation CAPSENSE™ technology has built in ability for multi-channel synchronization and CPU is not required for this. Multi-channel operation is an added advantage to support applications such as large touchpad, which require many sensor pins for interfacing. For example, a 6x8 touchpad can be configured as shown in [Figure 284](#). In this figure, the sensors shown in blue color is scanned by channel 0 (MSC0) and green color is scanned by channel 1 (MSC1).

5 PSoC™ 6 application notes

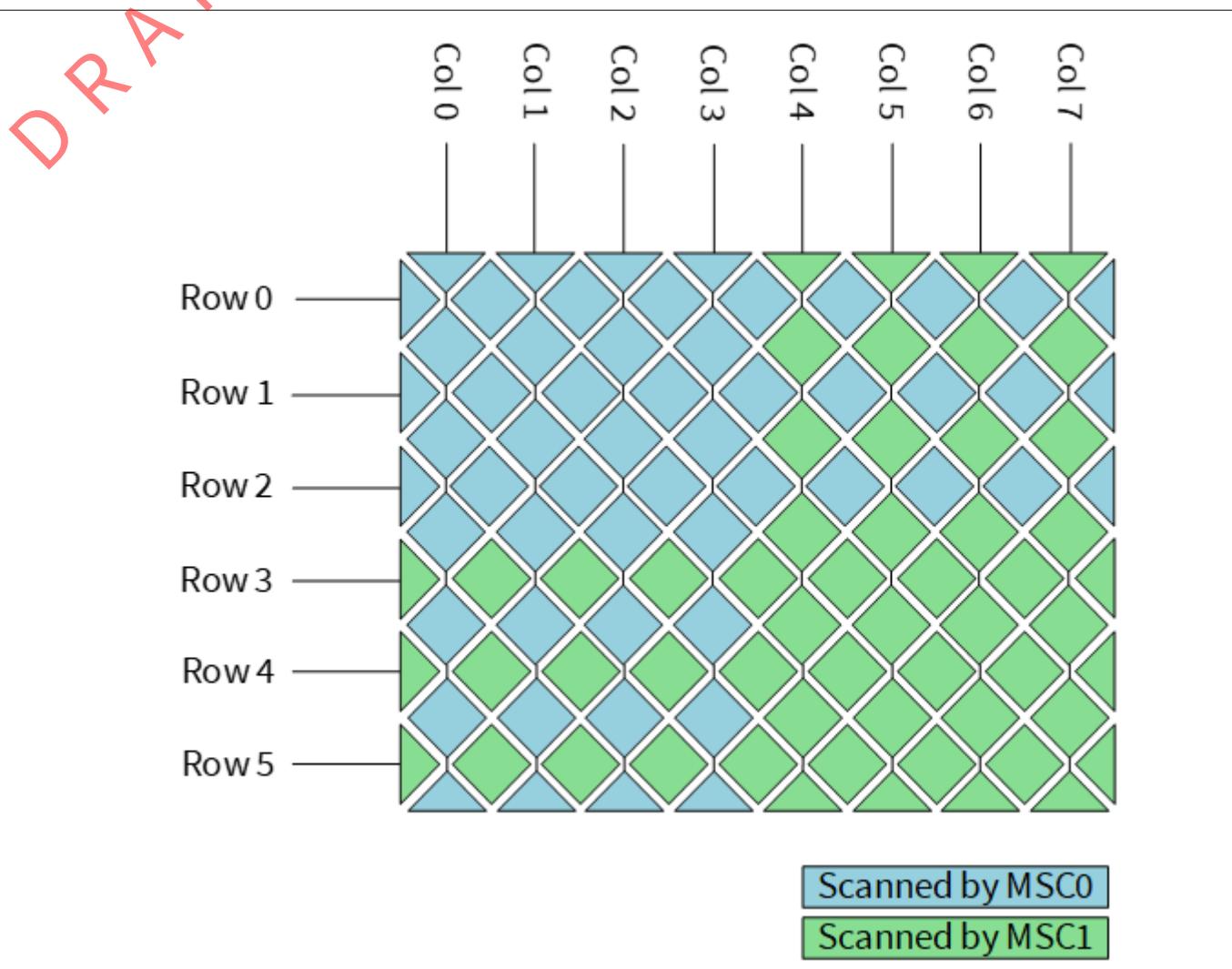


Figure 284 Scanning 6x8 CSD touchpad using multi-channel

In this case, channel 0 and channel 1 can scan one of its sensors at the same time. To avoid any cross-talk noise, sensors to be scanned together should be selected such that the physical distance between the two sensor is as maximum as possible, and should avoid combining row and column sensors.

In the above example, the recommended scan configuration is as shown in [Table 51](#). All the sensors that belong to same slot is scanned together.

Table 51 Channel scan configuration

Slot #	Channel 0 sensor	Channel 1 sensor
Slot 0	Col 0	Col 4s
Slot 1	Col 1	Col 5
Slot 2	Col 2	Col 6
Slot 3	Col 3	Col 7
Slot 4	Row 0	Row 3
Slot 5	Row 1	Row 4
Slot 6	Row 2	Row 5

~~5 PSoC™ 6 application notes~~

~~Button widget tuning~~

~~Button widget tuning~~ section provides high-level steps for tuning CSD button. The [CE231078 PSoC™ 4: MSC CAPSENSE™ CSD Button Tuning](#) explains tuning of self capacitance-based button widgets in the Eclipse IDE for ModusToolbox™. For details on the Component and all related parameters, see the [Component datasheet](#).

~~Slider widget tuning~~

~~Slider widget tuning~~ section provides high-level steps for tuning CSD slider. The [CE232776 PSoC™ 4: MSC CAPSENSE™ CSD slider tuning](#) explains tuning of self-capacitance-based slider widgets in the Eclipse IDE for ModusToolbox™. For details on the Component and all related parameters, see the [Component datasheet](#).

~~Touchpad widget tuning~~

~~Touchpad widget tuning~~ section provides high-level steps for tuning CSD-RM touchpad. The explains tuning of self-capacitance-based touchpad widgets in the Eclipse IDE for ModusToolbox™. For details on the Component and all related parameters, see the [Component datasheet](#).

Following are the basic rules for Scan Order tab for using CSD-RM Touchpad Widget on multi-channels:

1. Scanning in Fifth-Generation CAPSENSE™ is ordered using slot numbers. A single slot number can be assigned to one sensor in all the channels and scanning that particular slot, scans all the sensors in that slot in sync
2. For CSD-RM Touchpad same slot should only be assigned to the row or to the column. Thus, avoiding scanning of a row and column element together which will cause cross-talk.
3. Slot numbers should be assigned in such a way that there is a maximum distance between the sensors which is having same slot number
4. Should not mix CSD and CSX sensors in a single slot
5. Touchpad sensors should be equally divided between channels for optimizing scan duration
6. All channels must have equal number of sensors (scans) for "consensus" method to work. If number of sensor in each channel is not equal, "empty slots" are added to respective channels
7. Within a slot all sensors should have the same sense clock and same number of sub-conversions

[Figure 285](#) shows an example of slot configuration for an 8x6 CSD-RM touchpad.

The screenshot shows the 'Slot' configuration tab for a 'Touchpad0' component. The main table lists electrodes (Row0 to Row5) and their assignments across 8 columns (Col0 to Col6). The summary table on the right shows the distribution of slots across two channels (Ch 00 and Ch 01).

		Assign slots								Lock	
Touchpad0		Electrode	Col0	Col1	Col2	Col3	Col4	Col5	Col6	Ch 00	Ch 01
Channel	00	00	00	00	01	01	01	01	01	Slot 0	Slot 0
Ganged	No	No	No	No	No	No	No	No	No	Slot 1	Slot 1
Pin	P3[0]	P3[1]	P3[4]	P3[5]	P0[0]	P0[1]	P0[2]			Slot 2	Slot 2
Slot	Slot 0	Slot 1	Slot 2	Slot 3	Slot 0	Slot 1	Slot 2			Slot 4	Slot 4
Electrode	Channel	Ganged	Pin	Slot						Slot 5	Slot 5
Row0	00	No	P3[6]	Slot 4						Slot 6	Slot 6
Row1	00	No	P3[7]	Slot 5							
Row2	00	No	P4[4]	Slot 6							
Row3	01	No	P5[0]	Slot 4							
Row4	01	No	P5[3]	Slot 5							
Row5	01	No	P5[4]	Slot 6							

Slot	Ch 00	Ch 01
Slot 0	0	0
Slot 1	0	0
Slot 2	0	0
Slot 3	0	0
Slot 4	0	0
Slot 5	0	0
Slot 6	0	0

Figure 285 Slot configuration for an 8x6 CSD-RM touchpad

Proximity widget example

For tuning a proximity sensor, see [AN92239 - Proximity sensing with CAPSENSE™](#).

~~5 PSoC™ 6 application notes~~

~~5.8.5.3.5 CSX-RM sensing method (Fifth-generation)~~

This section explains the basics of manual tuning using CSX-RM sensing method for the Fifth-Generation devices. It also explains the hardware parameters that influence the manual tuning procedure.

~~Basics~~

Conversion gain and CAPSENSE signal

Conversion gain will influence how much signal count the system observes for a finger touch on the sensor. If there is more gain, the signal is higher, and a higher signal means a higher achievable [Signal-to-noise ratio \(SNR\)](#). Note that an increased gain may result in an increase in both signal and noise. However, if required, you can use firmware filters to decrease noise. For details on available firmware filters, see [Table 41](#).

Conversion gain in single CDAC

In a mutual-capacitance sensing system, the rawcount counter is directly proportional to the mutual-capacitance between the Tx and Rx electrodes, as [Equation 63](#) shows.

$$\text{Rawcount}_{\text{counter}} = G_{\text{CSX}} C_M$$

Equation 63 Raw count relationship to sensor capacitance

Where,

G_{CSX} = Capacitance to digital conversion gain of CAPSENSE™ CSX

C_M = Mutual-capacitance between the Tx and Rx electrodes

Figure 97 shows the relationship between raw count and mutual-capacitance of the CSX sensor. The tunable parameters of the conversion gain in [Equation 64](#) are C_{ref} , $TxClk_{\text{Div}}$ and N_{Sub} .

The approximate value of this conversion gain is:

$$G_{\text{CSX}} = \text{MaxCount} \frac{2}{C_{\text{ref}} T x Clk_{\text{Div}}}$$

Equation 64 Capacitance to digital converter gain

Where, $\text{MaxCount} = N_{\text{Sub}} * T x Clk_{\text{Div}}$

The equation for raw count in the single CDAC mode, according to [Equation 63](#) and [Equation 64](#) is shown in [Equation 65](#).

$$\text{Rawcount}_{\text{Counter}} = N_{\text{Sub}} \frac{2 \times C_M}{C_{\text{ref}}}$$

Equation 65 Single CDAC mode raw counts

Where,

N_{Sub} = Number of sub-conversions

$TxClk_{\text{Div}}$ = Tx clock divider

C_M = Sensor mutual-capacitance

RefCDACCode = Reference CDAC value

$C_{\text{lsb}} = 8.86 \text{ fF}$

The tunable parameters of the conversion gain are C_{ref} , $TxClk_{\text{Div}}$, and N_{Sub} .

5 PSoC™ 6 application notes

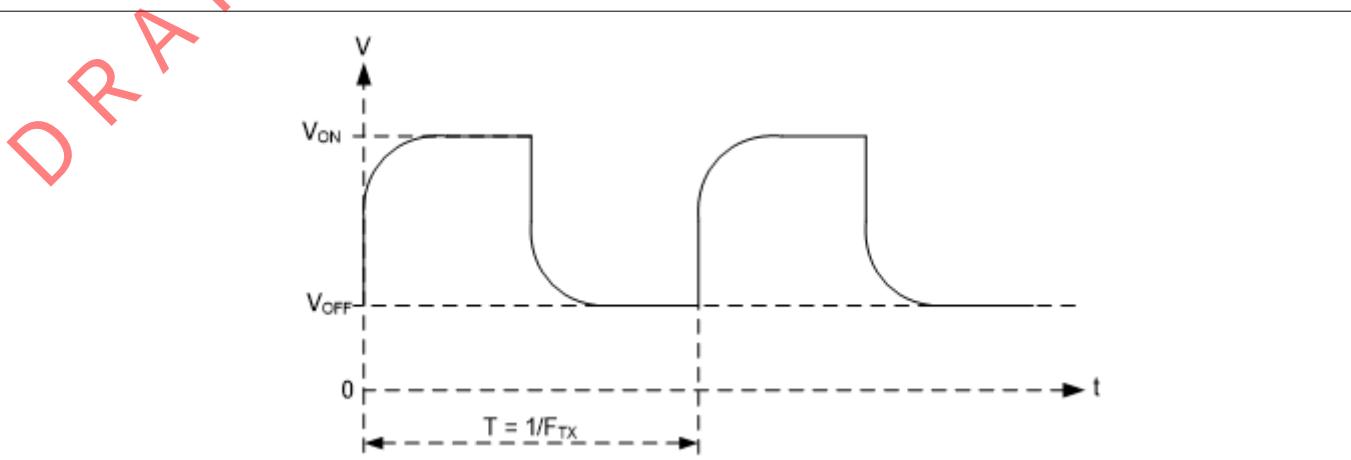


Figure 286 **Voltage at Tx node of the CSX sensor**

Note: The raw count observed from the Component is given by [Equation 66](#). See CAPSENSE™ CSX-RM sensing method (fifth-generation) for more details on Rawcount_{component}.

$$\text{Rawcount}_{\text{Component}} = \text{MaxCount} - \text{Rawcount}_{\text{Counter}}$$

Equation 66 **Rawcount_{component}**

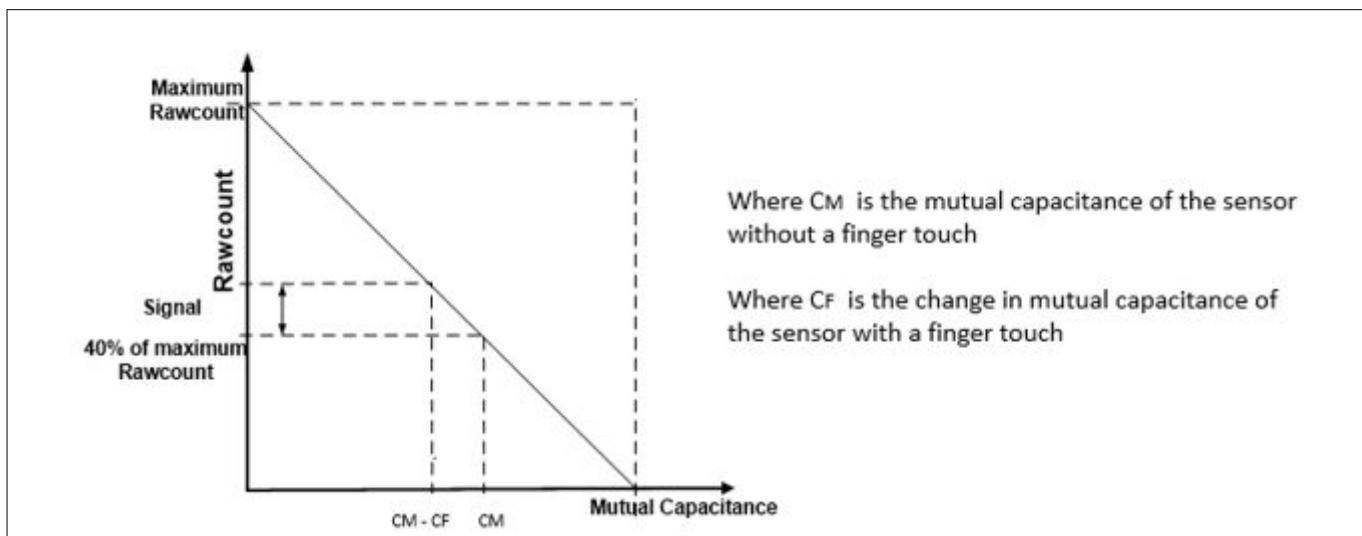


Figure 287 **Raw count vs sensor mutual-capacitance**

Conversion gain in dual CDAC mode

The equation for raw count in the dual CDAC mode, according to [Equation 23](#) and [Equation 63](#) is shown in [Equation 67](#).

$$\text{rawcount} = G_{\text{CSX}} C_M - \text{Maxcount} \frac{2 \times C_{\text{comp}}}{C_{\text{refComp}} \text{CLK}_{\text{div}}}$$

Equation 67 **Dual CDAC mode raw counts**

Where,

$\text{Maxcount} = N_{\text{Sub}} * SnsClk_{\text{Div}}$

$SnsClk_{\text{Div}}$ = Sense clock divider

~~5 PSoC™ 6 application notes~~

N_{Sub} = Number of sub-conversions

C_{ref} = Reference capacitance = $\text{RefCDACCode} * C_{\text{lsb}}$

C_{comp} = Compensation capacitance = $\text{CompCDACCode} * C_{\text{lsb}}$

$\text{CompCLK}_{\text{Div}}$ = CDAC compensation divider

C_M = Sensor mutual-capacitance

RefCDACCode = Reference CDAC value

CompCDACCode = Compensation CDAC value

$C_{\text{lsb}} = 8.86 \text{ fF}$

G_{CSX} is given by [Equation 64](#)[Equation 64](#).

Selecting CAPSENSE™ hardware parameters

CAPSENSE™ hardware parameters govern the conversion gain and CAPSENSE™ signal. [Table 52](#) lists the CAPSENSE™ hardware parameters that apply to the CSX-RM sensing method for the Fifth-Generation devices.

Table 52 CAPSENSE™ component hardware parameters

S. No	CAPSENSE™ parameter in ModusToolBox™
1	Tx clock divider
2	Tx clock source
3	Modulator clock divider
4	Reference CDAC value
5	CDAC compensation divider
6	Compensation CDAC value
7	Number of sub-conversions
8	Enable CDAC dither

Scan mode

Scan mode can be set as CS-DMA or Interrupt Driven mode. For autonomous scanning select DMA mode and for legacy interrupt based scanning select Interrupt Driven mode.

Sensor connection method

Autonomous scanning is only available in CTRLMUX method, but the numbers of supported pins are limited in this method (see the [Device datasheet](#) for supported pins). Additionally provides better immunity to on-chip IO noise. Choose AMUXBUS method to support more number of pins in Interrupt Driven mode.

In CTRLMUX connection method for CSX sensors, choose Inactive sensor connection as VDDA/2 and ensure to add empty scan slots before the first sensor scan for initializing the voltages on Rx lines to VDDA/2. See [Touchpad widget tuning](#) code examples for detailed steps on creating empty slots.

Modulator clock frequency

It is best to choose the highest allowed clock frequency for the given device because a higher modulator clock frequency leads to a higher sensitivity/signal, increased accuracy, and lower noise for a given C_M to digital count conversion as [Equation 63](#) and [Equation 64](#) indicates. Also, a higher value of $F_{\text{MOD}}/F_{\text{TX}}$ ensures lower width of [Flat-spots](#) in C_M to raw count conversion.

~~5 PSoC™ 6 application notes~~

~~Initialization sub-conversions~~

As part of initialization, C_{MOD} 's needs to be charged at required voltage ($VDDA/2$). There are three phases in initialization – C_{MOD} initialization, C_{MOD} short and initialization sub-conversions. During C_{MOD} initialization phase C_{MOD1} is pulled to GND and C_{MOD2} is pulled to $VDDA$. During C_{MOD} short phase both capacitors are tied together so the charge is shared to produce a voltage close to $VDDA/2$ on both. After the 2 phases the scanning is started but rawcount is discarded for number of init sub-conversions.

Number of init sub-conversions should be selected based on [Equation 68](#).

$$\text{Number of init subconversions} = \text{ceiling}\left(\frac{C_{MOD} \times V_{OS}}{2 \times VDDA \times C_M \times (1 - \text{Base \%}) \times (\frac{1}{\text{Bal \%}} - 1)}\right) + 1$$

or

$$\text{Number of inti subconversions} = \text{ceiling}\left(\frac{C_{MOD} \times V_{OS}}{VDDA \times TxClkDiv \times C_{ref} \times (1 - \text{Bal \%})}\right) + 1$$

Equation 68 Number of init sub-conversions

Where,

C_{MOD} = Modulator capacitor

V_{OS} = Comparator offset voltage (3 mV of PSoC 4100S Max device)

C_M = Sensor mutual-capacitance

Base% = Baseline compensation percentage

Bal% = Rawcount calibration percentage

C_{ref} = Reference capacitance = RefCDACCode * C_{lsb}

RefCDACCode = Reference CDAC value

C_{lsb} = 8.86fF

Tx clock parameters

There are two parameters that are related to the Tx clock: Sense clock source and Sense clock frequency.

Tx clock source

Select “Auto” as the clock source for the Component to automatically select the best Tx clock source between Direct and Spread Spectrum Clock (SSCx) for each widget. If “Auto” option is not selected, then choose the clock source based on the following:

- Direct – Clock signal with a fixed clock frequency. Use this option for most cases
- Spread spectrum clock (SSCx) – If you choose this option, the Tx clock signal frequency is dynamically spread over a predetermined range. Use this option for reduced EMI interference and avoiding Flat-spots. However, when selecting SSCx clock, ensure to select the Tx clock frequency, modulator clock frequency, and number of sub-conversion such that the conditions mentioned in [Component datasheet/ModusToolbox™ CAPSENSE™ configurator guide](#) for SSCx clock source selection are satisfied
- Pseudo Random Sequence (PRSx) – Use PRSx (pseudo random sequence) modes to remove flat-spots and improve EMI/EMC radiation and susceptibility. In 5th Generation CAPSENSE™, PRS clock introduces signal/sensitivity loss at higher rawcount calibration percent, hence 65% rawcount calibration is recommended when PRS clock is used

~~DETAILED~~ 5 PSoC™ 6 application notes

Tx clock frequency

The Tx clock frequency determines the duration of each sub-conversion as explained in the CAPSENSE™ CSX-RM sensing method (fifth-generation) section. The Tx clock signal must completely charge and discharge the sensor parasitic capacitance; and can be verified by checking the signal in an oscilloscope or it can be set using [Equation 63](#). In addition, ensure that the auto-calibrated CDAC code lies in the mid-range (for example, 6-200) for the selected . If the auto-calibrated CDAC code lies out of the recommended range, tune such that it falls in the recommended range and satisfies [Equation 69](#).

$$F_{TX} < \frac{1}{2 \times 5 \times R_{SeriesTx} C_{pTx}}$$

Equation 69 Condition for selecting Tx clock frequency

Where,

C_{pTx} = Tx electrode parasitic capacitance

$R_{SeriesTx}$ = Total series-resistance, including the $R_{internal}$ resistance of the internal switches, the recommended external series resistance of 2 KΩ (connected on PCB trace connecting sensor pad to the device pin), and trace resistance if using highly resistive materials (example ITO or conductive ink).

$R_{internal}$ = Internal resistance, this varies based on scan modes, see [Table 53](#).

Table 53 Internal resistance for sensor

Scan mode	$R_{internal}$
CTRLMUX	950 Ω
AMUXBUS	500 Ω

The value for C_p can be estimated using the CSD Built-in-Self-test APIs. See the [Component datasheet/middleware document](#) for details.

To minimize the scan time, as [Equation 70](#) shows, it is recommended to use the maximum Tx clock frequency available in the component drop-down list that satisfies [Equation 69](#).

$$T_{CSX} = \frac{N_{Sub}}{F_{TX}}$$

Equation 70 Scan time of CSX sensor

Where,

N_{Sub} = Number of sub-conversions

Additionally, if you are using the SSCx clock source, ensure that you select the Tx clock frequency that meets the conditions mentioned in [Component datasheet/middleware document/ModusToolbox™ CAPSENSE™ configurator guide](#) in addition to these conditions.

Number of sub-conversions

The number of sub-conversions decides the sensitivity of the sensor and sensor scan time. From [Equation 24](#) for a fixed modulator clock and Tx clock, increasing the number of sub-conversions (N_{Sub}) increases the signal and SNR. However, increasing the number of sub-conversions also increases the scan time of the sensor per [#unique_673/unique_673_Connect_42_equation-figure_ptc_kn5_2tb](#).

Initially, set the value to a low number (for example, 20), and use the Tuner GUI to find the SNR of the sensor. If the SNR is not > 5:1 with the selected N_{Sub} , increase then N_{Sub} in steps such that the SNR requirement is met.

~~5 PSoC™ 6 application notes~~

~~Capacitive DACs~~

CSX-RM in Fifth-Generation supports two CDACs: Reference CDAC (C_{ref}) and Compensation CDAC (C_{comp}) that balance C_{MOD} 's as Figure 49 shows. These govern the Conversion gain and CAPSENSE™ signal for capacitance-to-digital conversion. The CAPSENSE™ Component allows the following configurations of the CDACs:

- Enabling or disabling of Compensation CDACs
- Enabling or disabling of Auto-calibration for the CDACs
- Compensation CDAC divider, DAC code selection for Reference and Compensation CDACs if auto-calibration is disabled

Reference CDAC (C_{ref})

The reference CDAC is used to compensate the charge transferred by the sensor mutual-capacitance (C_M) from C_{MOD} . The number of times it is switched depends on the mutual-capacitance of sensor.

C_{ref} should satisfy below critieria:

- For Compensation Disabled:
 $RefCDACCode \geq 6$
- For Compensation Enabled:

$$RefCDACCode \geq \frac{10}{CDAC Compensation Divider}$$

Where,

C_{ref} = Reference capacitance = $RefCDACCode * C_{lsb}$

$RefCDACCode$ = Reference CDAC value

$C_{lsb} = 8.86\text{fF}$

Compensation CDAC (C_{comp})

Enabling the compensation CDAC is called “dual CDAC” mode, and results in increased signal as explained in [Conversion gain and CAPSENSE signal](#). Enable the compensation CDAC for most cases.

The compensation capacitor is used to compensate excess mutual-capacitance from the sensor to increase the sensitivity. The number of times it is switched depends on the amount of charge the user application is trying to compensate from the sensor mutual-capacitance.

C_{comp} should satisfy below criteria:

- If $RefCDACCode = 1$, then $CompCDACCode \geq 98$

Where,

C_{comp} = Compensation capacitance = $CompCDACCode * C_{lsb}$

$CompCDACCode$ = Compensation CDAC value

$C_{lsb} = 8.86\text{ fF}$

Compensation CDAC divider

The number of times the compensation capacitor is switched in a single sense clock is denoted by K_{comp} . Select CDAC compensation divider based on below [Equation 71](#) such that below criteria is satisfied:

1. CDAC compensation divider ≥ 4
2. K_{comp} should be a whole number

$$\text{CDAC compensation divider} = \frac{\text{Tx clock divider}}{K_{comp}}$$

Equation 71 CDAC compensation divider

5 PSoC™ 6 application notes

~~Auto-calibration~~

This feature enables the firmware to automatically calibrate the CDAC to achieve the required calibration target of 40%. It is recommended to enable auto-calibration for most cases. Enabling this feature will result in the following:

Fixed raw count calibration to 40% of max raw count even with part-to-part C_M variation

Decrease the effect of [Flat-spots](#)

Automatically selects the optimum gain

For proper functioning of CAPSENSE™ under diverse environmental conditions, it is recommended to avoid very low or high CDAC codes. You can use CAPSENSE™ tuner to confirm that the auto-calibrated CDAC values fall in this recommended range. If the CDAC values are out of the recommended range, based on [Equation 63](#), [Equation 64](#), and [Equation 66](#), you may change the Calibration level or F_{mod} or F_{SW} to get the CDAC code in proper range.

Selecting CDAC codes

This is not the recommended approach. However, this could be used only if you want to disable auto-calibration for any reason. To get the CDAC code, you may first configure CAPSENSE™ Component with auto-calibration enabled and all other hardware parameters the same as required for final tuning and read back the calibrated CDAC values using [Tuner GUI](#). Then, re-configure the CAPSENSE™ Component to disable auto-calibration and use the obtained CDAC codes as fixed DAC codes read-back from the [Tuner GUI](#).

CDAC dither

As the input capacitance is swept the raw count should increase linearly with capacitance. There are regions where the raw count does not change linearly with input capacitance these are called flat-spots, see section [Flat-spots](#) for more details. Dithering helps to reduce flat-spots using a dither CDAC. The dither CDAC adds white noise that moves the conversion point around the flat region.

Selecting CAPSENSE™ software parameters

CAPSENSE™ software parameters in Fifth-Generation are the same as that for Fourth-Generation; therefore, these parameters could be selected as mentioned in the [Selecting CAPSENSE™ software parameters](#) section.

Configuring autonomous scan

Configuring autonomous scan in CSX-RM sensing is the same as that for CSD-RM sensing; therefore, configurate autonomous scan as mentioned in the [Configuring autonomous scan](#) section.

Multi-channel scanning

Multi-channel scanning in CSX-RM sensing is the same as that for CSD-RM sensing; therefore, refer Multi-channel scanning [Chapter](#) for more details.

Button widget tuning

Button widget tuning section provides high-level steps for tuning CSX button. The [CE231079 PSoC™ 4: MSC CAPSENSE™ CSX button tuning](#) explains tuning of mutual-capacitance based button widgets in the Eclipse IDE for ModusToolbox™. For details on the Component and all related parameters, see the [Component datasheet](#).

~~5 PSoC™ 6 application notes~~

~~Touchpad widget tuning~~

~~Touchpad widget tuning~~ section provides high-level steps for tuning the CSX Touchpad. The [CE232275 PSoC™ 4: MSC multi-touch mutual-CAPSENSE™ touchpad tuning](#) explains tuning of mutual-capacitance based button widgets in the Eclipse IDE for ModusToolbox™. For details on the Component and all related parameters, see the [Component datasheet](#).

Rules for Scan Order tab for CSX widget when multi-channels are enabled:

1. Scanning in Fifth-Generation CAPSENSE™ is ordered using slot numbers. A single slot number can be assigned to one sensor in all the channels and scanning that particular slot, scans all the sensors in that slot in sync
2. Slot numbers should be assigned in such a way that there is a maximum distance between the Rx electrode which is having same slot number, thus avoiding any potential cross-talk
3. Tx and Rx electrode of a sensor can be assigned to two different channels or same channel. The sensor belongs to the channel which sensor Rx electrode is connected
4. Rx electrodes should be equally divided between channels for optimizing scan duration
5. Any of the channel can generate Tx signal for all channels
6. Tx electrodes can be assigned in any order between channels
7. All channels must have equal number of sensors (scans) for “consensus” method to work. If number of scans in each channel is not equal, “empty slots” are added to respective channels.
8. Should not mix CSD and CSX sensors in a single slot
9. Within a slot all sensors should have the same sense clock and same number of sub-conversions
10. [Figure 288](#) shows an example of slot configuration for an 8x6 CSX-RM touchpad

The screenshot shows the 'Slot configuration for 8x6 CSX-RM touchpad' window. It contains two main sections: 'Electrode' and 'Capacitors'. The 'Electrode' section has tables for 'Rx0' through 'Rx7' and 'Tx0' through 'Tx5'. The 'Capacitors' section has tables for 'Cmod1' and 'Cmod2' across 'P4[0]' through 'P7[1]'. To the right, a 'Summary' table shows the assignment of slots to channels (Ch 00 and Ch 01). The 'Summary' table has rows for Slot 0 through Slot 18, with each row containing 'Ch 00' and 'Ch 01' columns. Most slots are assigned to both channels, while some are assigned to only one.

Slot	Ch 00	Ch 01
Slot 0	0	0
Slot 1	0	0
Slot 2	0	0
Slot 3	0	0
Slot 4	0	0
Slot 5	0	0
Slot 6	0	0
Slot 7	0	0
Slot 8	0	0
Slot 9	0	0
Slot 10	0	0
Slot 11	0	0
Slot 12	0	0
Slot 13	0	0
Slot 14	0	0
Slot 15	0	0
Slot 16	0	0
Slot 17	0	0
Slot 18	0	0

Figure 288 Slot configuration for 8x6 CSX-RM touchpad

5.8.5.3.6 Manual tuning trade-offs

When manually tuning a design, it is important to understand how the settings impact the characteristics of the capacitive sensing system. Any CAPSENSE™ design has three major performance characteristics: reliability, power consumption, and response time.

- **Reliability** defines how CAPSENSE™ systems behave in adverse conditions such as a noisy environment or in the presence of water. High-reliability designs will avoid triggering false touches, and ensure that all intended touches are registered in these adverse conditions

Power consumption is defined as the average power drawn by the device, which includes, scanning, processing, and low-power mode transitions as explained in [Low-power design](#). Quicker scanning and

~~DRAFT~~ 5 PSoC™ 6 application notes

processing of the sensors ensures that the device spends less time in a higher power state and maximizes the time it can spend in a lower power sleep state.

- **Response time** defines how much time it takes from the moment a finger touches the sensor until there is a response from the system. Because the lowest response time is limited by the scan and processing time of the sensors, it is important to properly define and follow a timing budget. A good target for total response time is below 100 ms

These performance characteristics depend on each other. The purpose of the tuning process is to find an optimal ratio that satisfies the project's specific requirements. When planning a design, it is important to note that these characteristics usually have an inverse relationship. If you take action to improve one characteristic, the others will degrade.

For example, if you want to use CAPSENSE™ in a toy, it is more important to have a quick response time and low power consumption. In a different example, such as a "Start/Stop" button for an oven, reliability is the most important characteristic and the response time and power consumption are secondary.

Now let us consider the factors that affect reliability, power consumption, and response time.

Figure 289 shows dependencies between CAPSENSE™ characteristics, measurable parameters, and actual CAPSENSE™ configurable parameters.

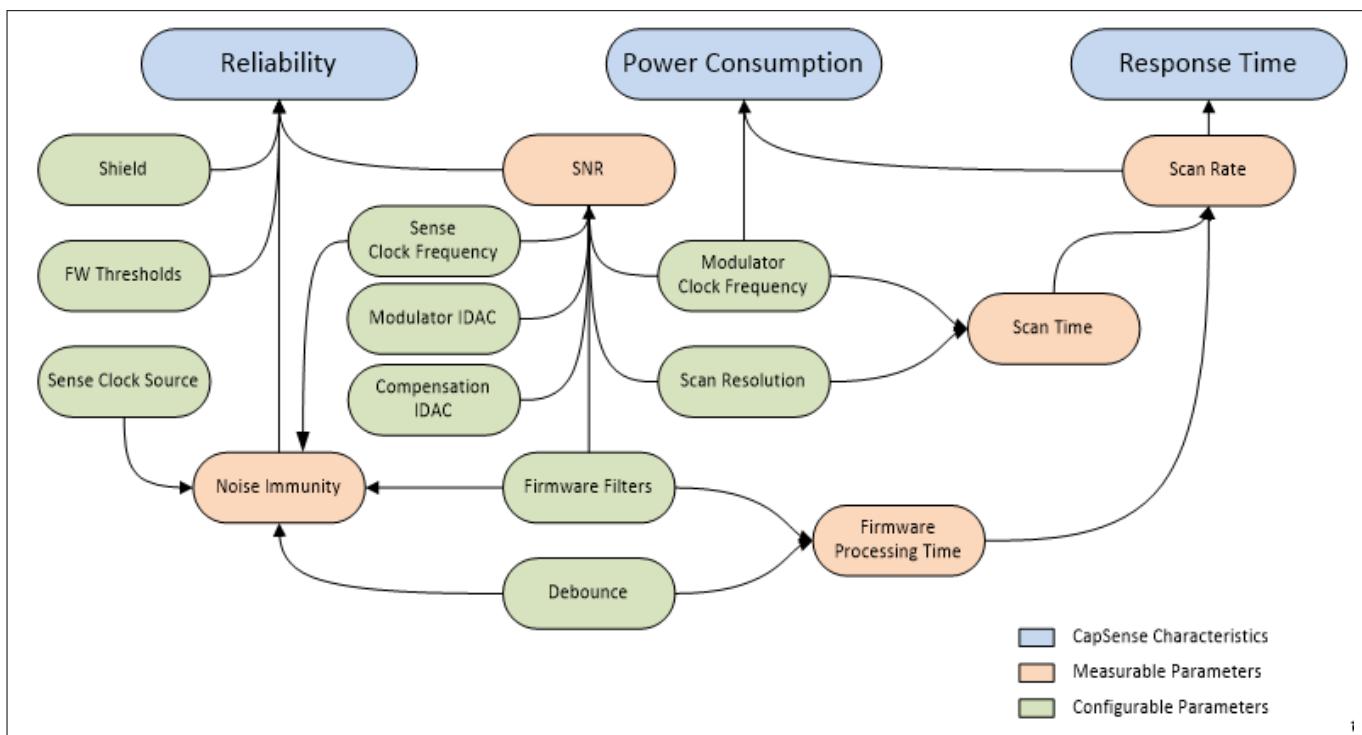


Figure 289 CAPSENSE™ parameter relationships

Reliability

The following factors affect reliability:

1. **Signal-to-noise ratio (SNR):** SNR gives a measure of confidence in a valid touch signal. For reliable CAPSENSE™ operation, it should be greater than 5. Manual tuning can ensure optimal SNR in specific designs
2. **Noise immunity:** It is the ability of the system to resist external or internal noise. Typical examples of external noise are ESD events, RF transmitters such as Bluetooth® LE, switching relays, power supply, and so on. The internal noise source could be an LED driven by PWM, or I²C, or SPI communications for example. Even designs with good SNR may suffer from poor performance because of poor noise

5 PSoC™ 6 application notes

immunity. Manual tuning allows to tune frequencies and parameters to help avoid noise interference by allowing more control over selection of different parameters

Power consumption and response time

The following factors affect the power consumption and response time:

1. Scan rate: Scan rate can be defined as the frequency at which you scan the sensor. Scan rate decides the minimal possible time from the finger touch until it is reported. The maximum scan rate will be limited by the [Sensor scan](#)
2. Scan time: It is the time taken to scan and process a particular sensor. It affects power consumption as indicated in [Low-power design](#) and scan rate as indicated above. Manual tuning can achieve specific scan durations while maintaining a minimum SNR
3. Firmware touch delay: This can be caused by the [Debounce](#) procedure or use of Raw Data Noise Filters depending on the CAPSENSE™ component version you are using). Both affect scan time by adding to the processing time of a sensor and delay the touch reporting until a certain number of samples in a row show the touch signal. In both cases response rate is reduced, but reliability is usually improved

5.8.5.3.7 Tuning debug FAQs

This section lists the general debugging questions on CAPSENSE™ Component tuning. Jump to the question you have, for quick information on possible causes and solutions for your debugging topic.

The tuner does not communicate with the device

Cause 1: Your device is not programmed.

Solution 1: Make sure to [Program](#) your device with your latest project updates before launching the tuner.

Cause 2: The tuner configuration setting does not match the SCB Component setting.

Solution 2: Open the EzI2C slave component configuration window, that is, the Configure ‘SCB_P4’ dialog and verify that the settings match the configuration of the Tuner Communication Setup dialog. See the [CAPSENSE™ Component datasheet](#) for details on tuner usage.

Cause 3: Your I2C pins are not configured correctly.

Solution 3: Open the .cydwr file in Workspace Explorer and ensure the pin assignment matches what is physically connected on the board.

Cause 4: You did not include the CAPSENSE™ TunerStart API or another required tuner code.

Solution 4: Add the tuner code listed in [CAPSENSE™ Component datasheet](#) to your main.c and reprogram the device.

I am unable to update parameters on my device through the tuner

Cause 1: Your communications settings on the device are incorrect.

Solution 1: Review and make sure the settings in the UART/EZI2C configurator dialog and Tuner Communication Setup dialog match. Make sure that the sub-address size is equal.

I can connect to the device but I do not see any raw counts

Cause 1: You did not add the tuner code to your project.

Solution 1: Review the [Tuner GUI](#) section and add the tuner code to your main.c and reprogram the device.

5 PSoC™ 6 application notes

~~DRAFT~~
Difference counts only change slightly (10 to 20 counts) when a finger is placed on the sensor

Cause 1: The gain of your system is too low.

Solution 1: Review the [Tuner GUI](#) section of this document.

Cause 2: Your sensor parasitic capacitance is very high.

Solution 2: To confirm this issue, use the Built-in Self-Test (BIST) APIs documented in the [Component datasheet](#). These functions allow you to read out an estimate of the sensor parasitic capacitance. You can also confirm this reading independently with an LCR meter.

If your hardware has an option to enable [Driven shield signal and shield electrode](#), use this option in the advanced settings of the CAPSENSE™ Component configuration window. A driven shield around the sensors helps reduce the parasitic capacitance. When you enable this option, you may want to enable driving the shield to unused sensors by also changing the “Inactive Sensor connection” setting to “shield” in the advanced settings. If after enabling the shield, your C_P remains greater than the supported range of parasitic capacitance by the PSoC™ device, review your board layout to reduce C_P further, by following the PCB layout guidelines, and/or contact [Technical support](#) to review your layout. See [Component datasheet/middleware document](#) for more details on the supported range of C_P .

Cause 3: Your overlay may be too thick.

Solution 3: Review your [Overlay thickness](#) with respect to your [Overlay thickness](#).

Cause 4: Raw counts may be too close to saturation and hence, saturating when sensor is touched.

Solution 4: Tune IDAC to ensure that raw counts are tuned to ~85 percent of the max raw count for a given sensor according to the [Modulation and compensation IDACs](#) section.

After tuning the system, I see large amount of radiated noise during testing

Cause 1: The sense clock frequency is causing radiated noise in your system.

Solution 1: Reduce the sense clock frequency or enable PRS for your sensor based on [Electromagnetic compatibility \(EMC\) considerations](#) section. If it is already enabled, see the [Electromagnetic compatibility \(EMC\) considerations](#) section.

Cause 2: Large shield electrode may be contributing to a large radiated noise.

Solution 2: Reduce the size of shield electrode based on [Layout guidelines for liquid tolerance](#).

My scan time no longer meets system requirements after manual tuning

Cause: The noise and C_P of your system are high, which requires more scan time and filtering to achieve reliable operation.

Solution: C_P needs to be reduced. First, enable the [Driven shield signal and shield electrode](#) in the advanced settings of the CAPSENSE™ Component configuration window and ensure gain is set as high as possible by reviewing the [PCB layout guidelines](#). If your system still cannot meet final requirements, you may need to change your board layout to reduce C_P further, review the [PCB layout guidelines](#) for the same.

I am unable to calibrate my system to 85 percent

Cause 1: Your sensor may have a short to ground.

Solution: First, use a multimeter to check if there is a short between your sensor and ground. If it is present, review your schematic and layout for errors.

Cause 2: Your sensor C_P may be too high or too low.

Solution: If your hardware has an option to enable [Driven shield signal and shield electrode](#), use this option in the advanced settings of the CAPSENSE™ Component configuration window. A driven shield around the sensors

~~5 PSoC™ 6 application notes~~

helps reduces the parasitic capacitance. If you do not have a hardware option to use shield or if after enabling the shield, your C_P remains greater than the device supported C_P , contact [Technical support](#) to review your layout or for further application-specific guidance.

If you suspect the capacitance to be low compared to the minimum supported parasitic capacitance by the device, add a footprint of the capacitor to a pin. In the final design, if the C_P is identified to be lower than the supported range, place an additional compensation capacitor to increase the sensor C_P to the supported range by dynamically connecting it to the sensor while scanning. See the [Component datasheet/middleware document](#) to understand how to gang the sensors to an external compensation capacitor connected to a pin to increase the C_P whenever required.

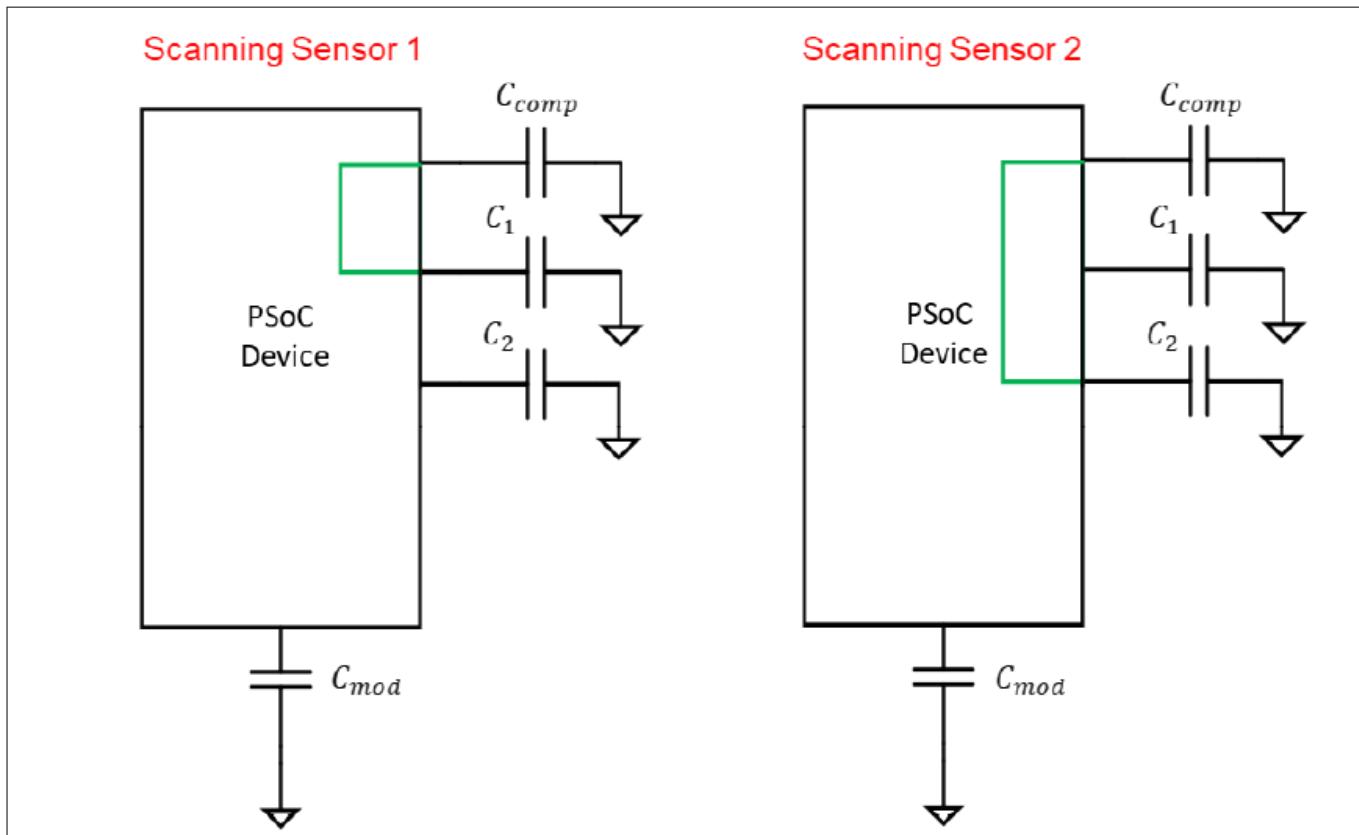


Figure 290 **Gang the sensors to the external compensation capacitor**

My slider centroid response is non-linear

Cause: Layout may not meet hardware design guidelines to ensure proper linearity.

Solution: Check the C_P of the sensors using the built-in self-test option in the General tab of the CAPSENSE™ configuration window and update the layout according to the [Slider design](#) section. See the [Component datasheet/middleware document](#) section for details on BIST API.

My slider segments have a large variation of CP

Cause: Your layout design caused your sensors to have an unbalanced C_P .

Solution: Your layout needs to be updated. Review [Slider design](#) and update your layout as required. If this is not immediately possible, you should re-tune every sensor to have a similar response. This will be a long iterative process and the preferred method is to update the hardware, if possible.

~~DETAILED~~ 5 PSoC™ 6 application notes

Raw counts show a level-shift or increased noise when GPIOs are toggled

Cause 1: The sensor traces are routed parallel to the toggling GPIOs on your PCB.

Solution: Your layout needs to be updated. Review [Trace routing](#) and update your layout as required. If the layout cannot be modified at the current stage, you could evaluate the use of firmware filters to reduce the peak-to-peak noise and hence improve SNR.

Cause 2: A large amount of current is being sunk through the GPIOs.

Solution: Limit the amount of DC current sink through the GPIOs when CAPSENSE™ sensors are being scanned. See [Schematic rule checklist](#). If the current sink through GPIOs is firmware-controlled, and the raw count-level-shift caused by current sink has a large difference compared to the touch signal, you could implement firmware techniques like resetting or re-initializing the CAPSENSE™ baseline whenever the current sink is enabled through the GPIOs. The baseline of the CAPSENSE™ sensor could be reset by using the `CapSense_InitializeWidgetBaseline()` API function as shown below:

```
CapSense_InitializeWidgetBaseline(CapSense_CSD_BUTTON_WDGT_ID);
```

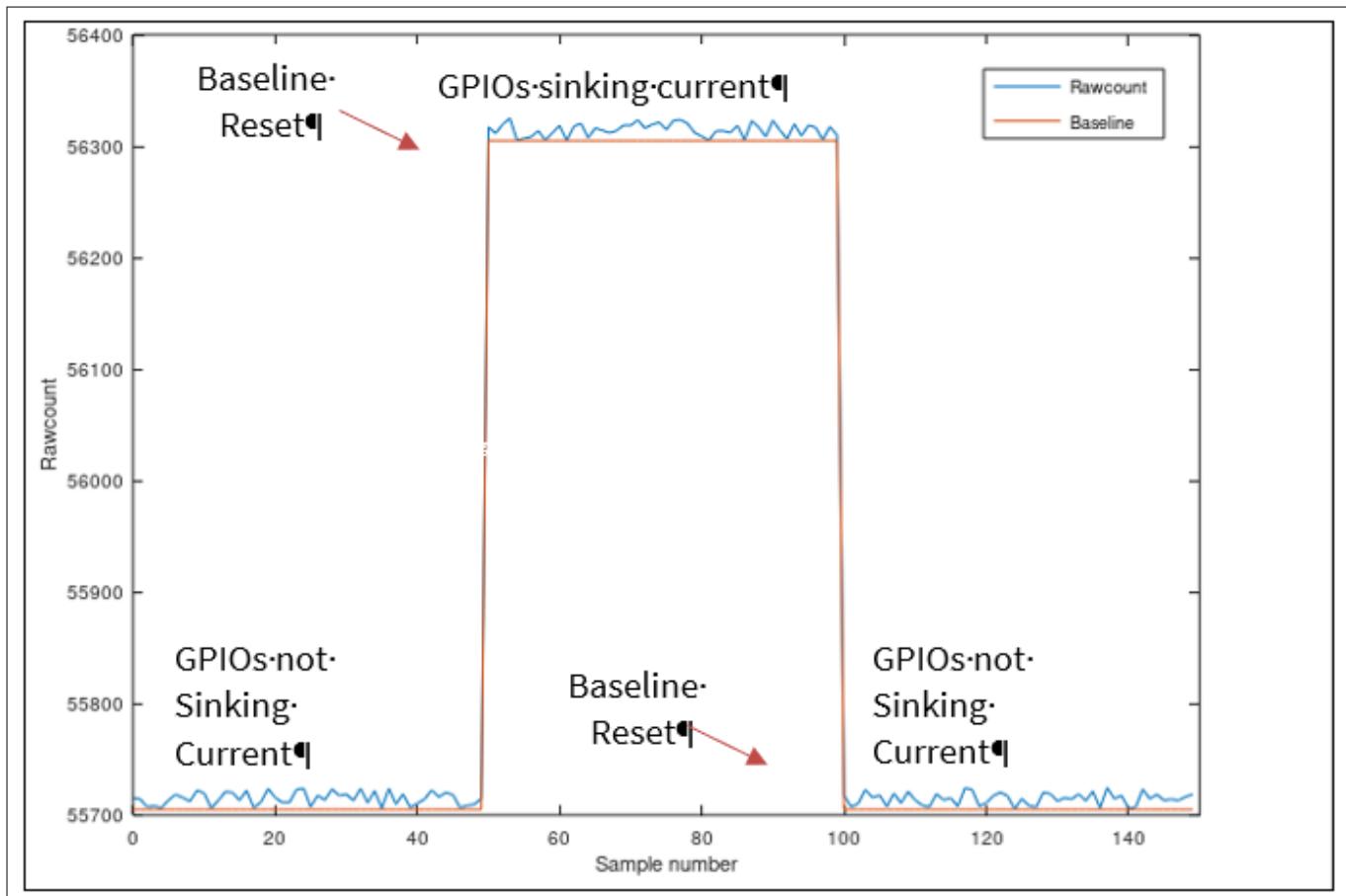


Figure 291 Resetting baseline using firmware technique

Cause 3: You did not follow the guidelines mentioned in [Sensor pin selection](#) section.

Solution: Follow the recommendations in [Sensor pin selection](#) section. In addition, for PSoC™ 6 family of devices, follow these guidelines on drive mode strength, switching frequency and slew rate selection, and so on:

- Reduce the drive strength of the switching GPIOs. [Table 54](#) lists the available drive strength options for the GPIOs. [Figure 292](#) shows an example on how to select the drive strength of the GPIOs using the [Device configurator](#) in the ModusToolbox™ project

5 PSoC™ 6 application notes

Table 54 Drive strength for GPIOs

Drive strength	Drive current in mA
Full	8
$\frac{1}{2}$	4
$\frac{1}{4}$	2
$\frac{1}{8}$	1

- Decrease the switching frequency of the GPIO being toggled
- Use GPIO slew rate as SLOW mode (note that this limits the toggling frequency to 1.5 MHz). See [Table 72](#) for more details
- Use PRS as the [Sense Clock](#) source.
- If possible, reduce VDDA to lower than 2.7 V
- Try to restrict GPIO switching to intervals between CAPSENSE™ scans

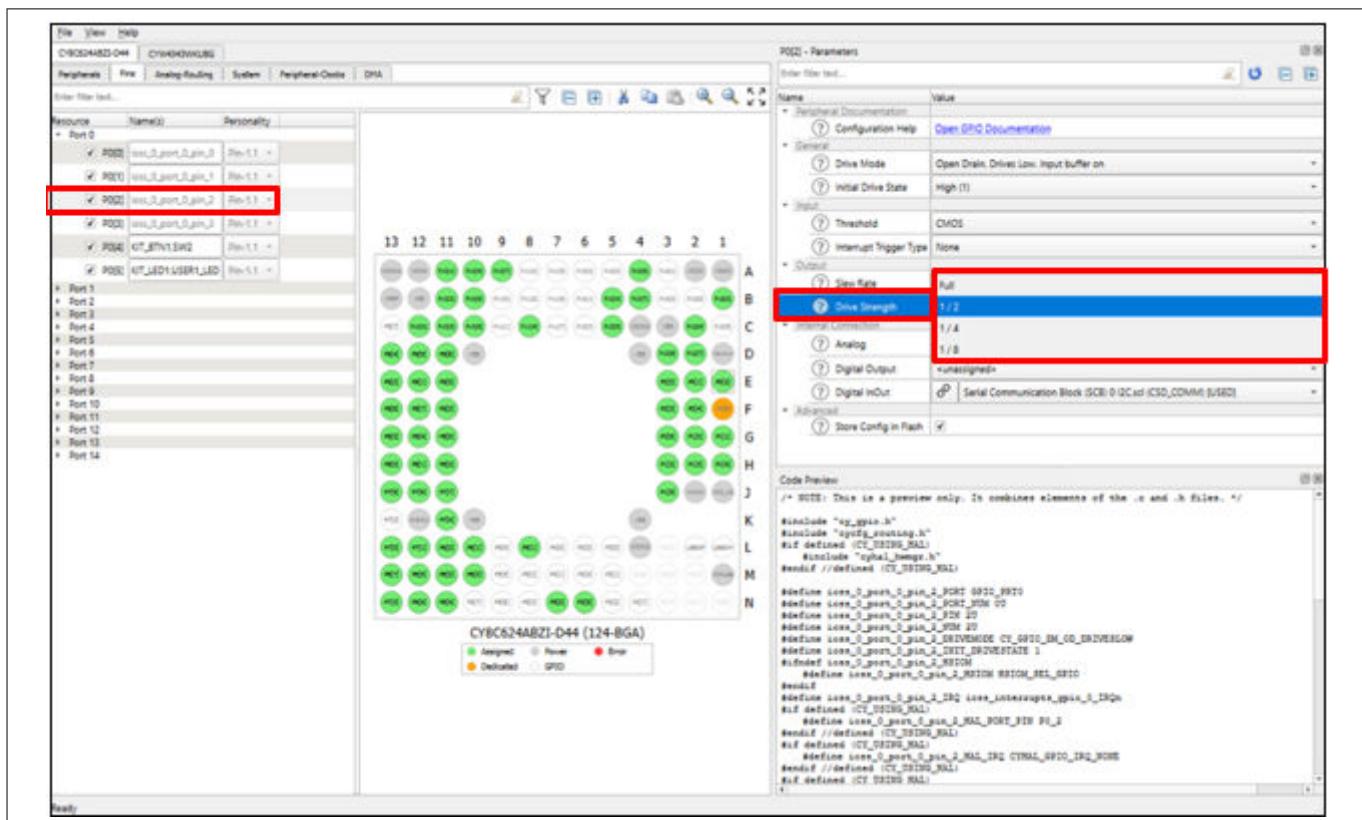


Figure 292 Selecting drive strength for GPIOs

I am getting a low SNR

Cause 1: Sensor is not tuned properly.

Solution: Follow the tuning guidelines in [CAPSENSE™ performance tuning](#).

Cause 2: CAPSENSE™ and other peripherals are not properly assigned to the recommended pin.

Solution: See [Sensor pin selection](#) and [Raw counts show a level-shift or increased noise when GPIOs are toggled](#) for more details.

5 PSoC™ 6 application notes

~~DO NOT USE~~ Cause 3: HFCLK source may be causing higher noise for a PSoC™ 6 device.

Solution: For the best performance of CAPSENSE™ in PSoC™ 6 family of devices, use HFCLK derived from the IMO/ECO+PLL clock source. This clock source provides the best SNR performance. [Figure 293](#) shows how to change the clock settings using the System tab in the [Device configurator](#) for a ModusToolbox™ project. See [AN221774 - Getting started with PSoC™ 6 MCU](#) for more details on changing the device clock.

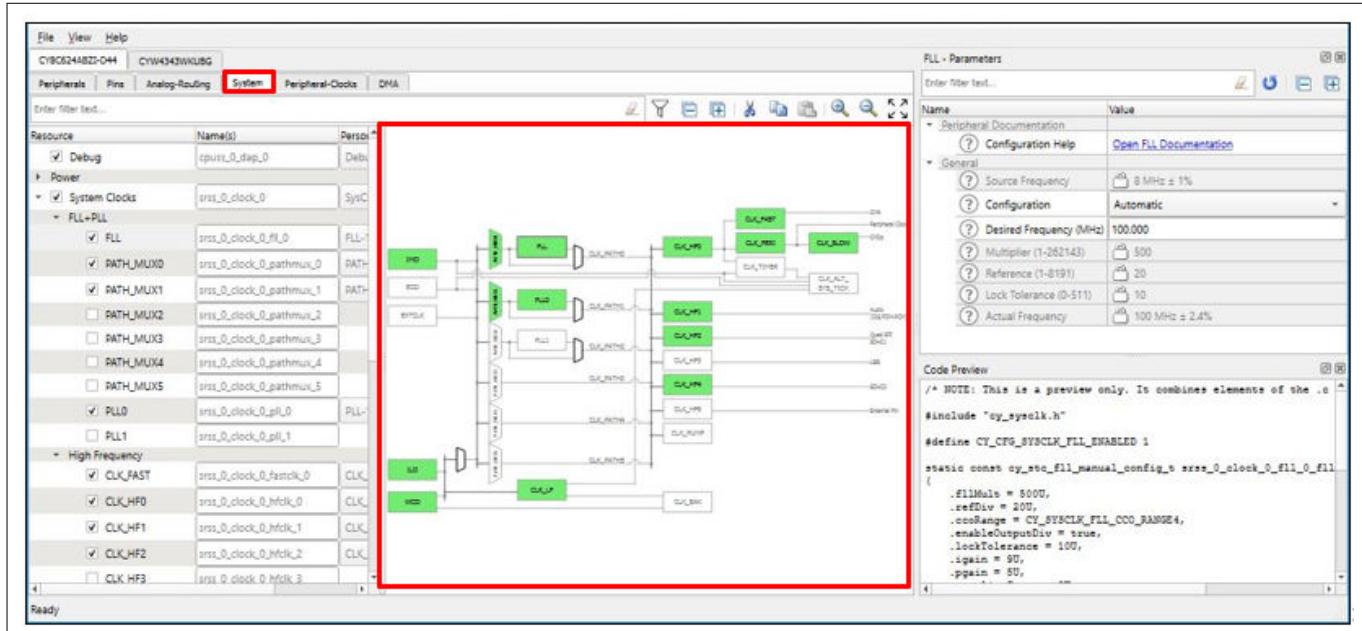


Figure 293 Changing clock settings in device configurator

I am observing a low CM for my CSX button

Cause: The mutual capacitance between the Tx and Rx electrode should be higher than approximately 750 fF for proper IDAC calibration.

Solution: It is recommended to have two free pins in your device with footprint to add extra Capacitance if C_M of the button turn out to be low. We could then increase the sensor C_M to the supported range by dynamically connecting external capacitor to the CSX sensor while scanning as shown in the below figure, where Pin1 is ganged to the Tx pin and Pin2 is ganged to the Rx pin of the sensor respectively. This will help in mitigating low C_M risk if it is found during testing phase. See [Component datasheet/middleware document](#) to understand how to gang the sensor.

[Figure 294](#) shows the addition of the external capacitor as a button widget in the CAPSENSE™ component and assigning dedicated pins to the Tx and Rx electrode. [Figure 295](#) shows the ganging of the sensor to the external capacitor by assigning Selected pins to both sensor pin and external capacitor pin, this must be done for both Rx and Tx electrode. There is no need to scan the external capacitor while scanning of the widgets, thus we can selectively scan widgets using the APIs `CapSense_SetupWidget()` and `CapSense_Scan()` provided by the CAPSENSE™ component.

5 PSoC™ 6 application notes

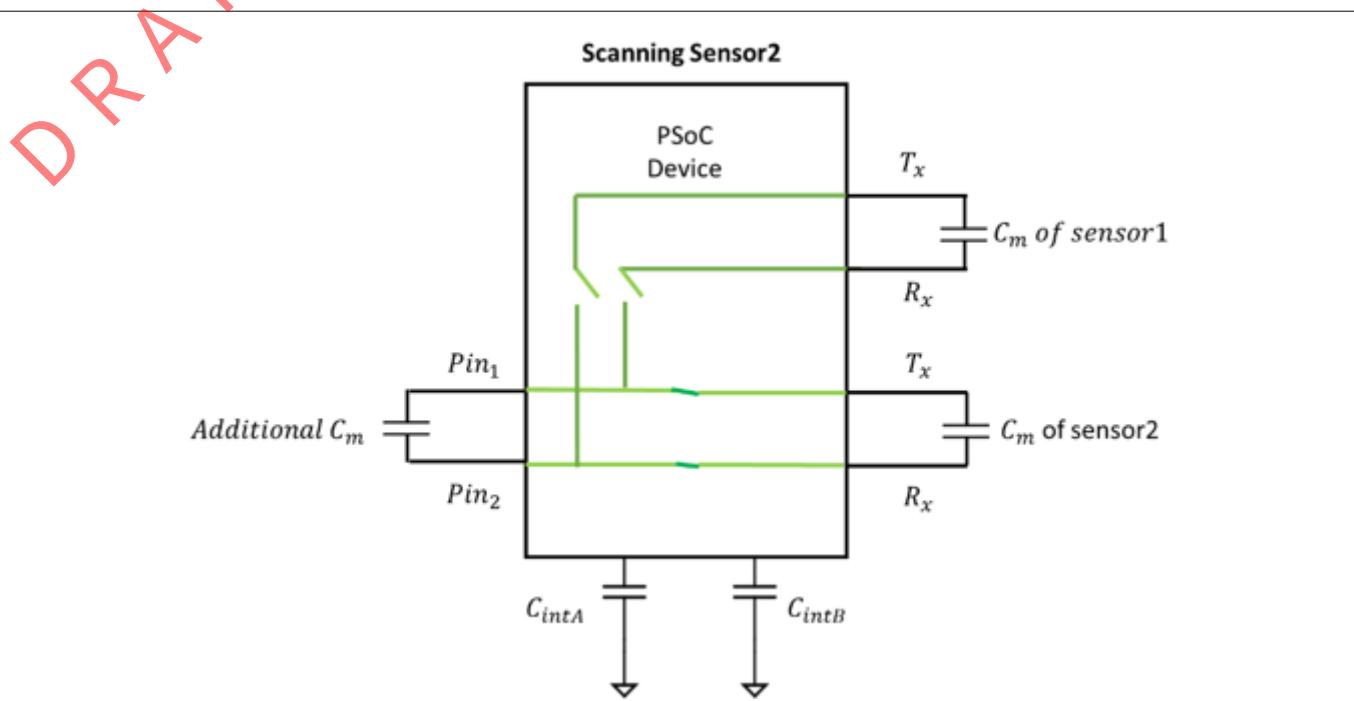


Figure 294 Ganging external capacitor to increase the C_M of the sensor

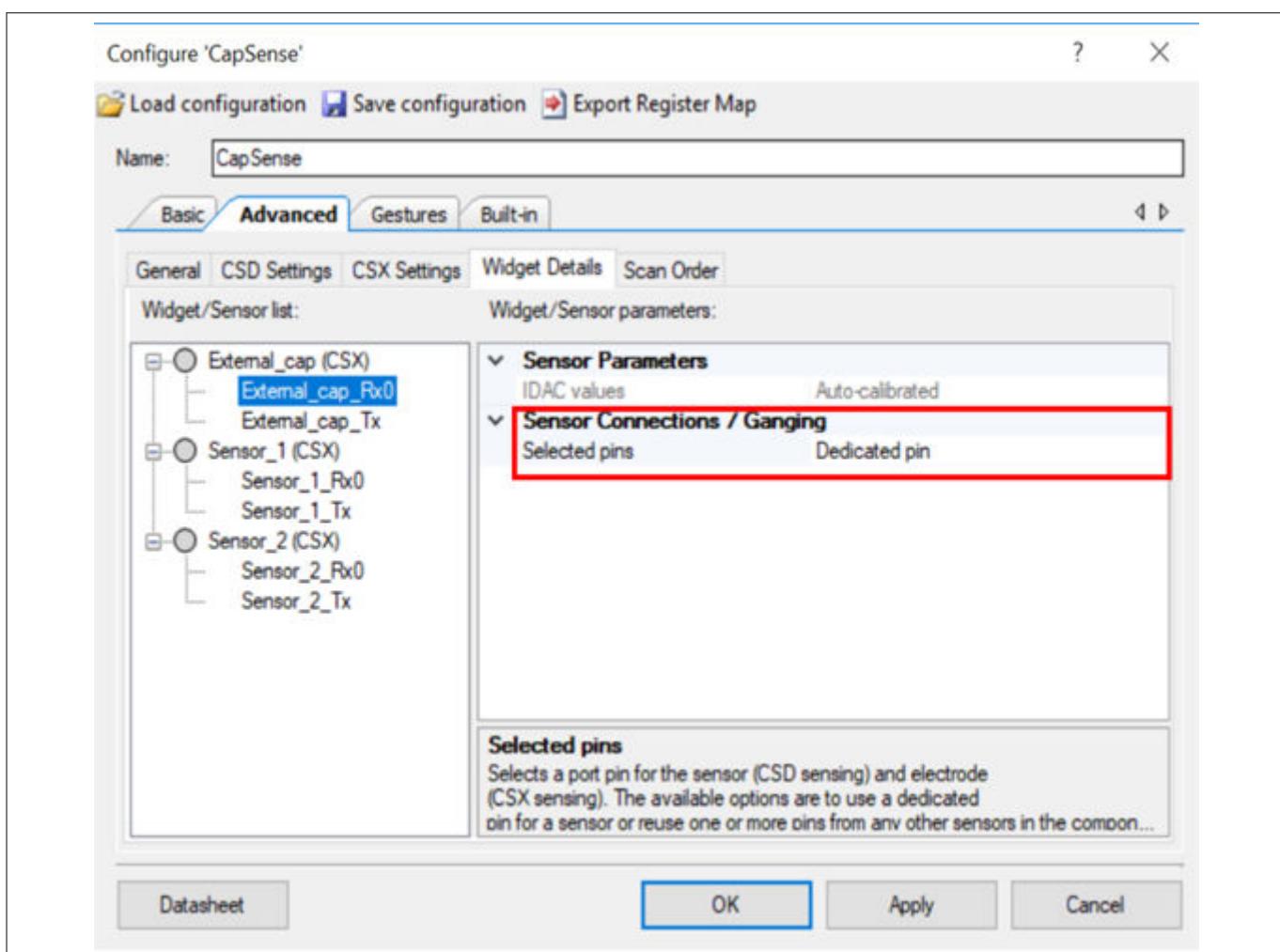


Figure 295 Assigning dedicated pins to the external capacitors

5 PSoC™ 6 application notes

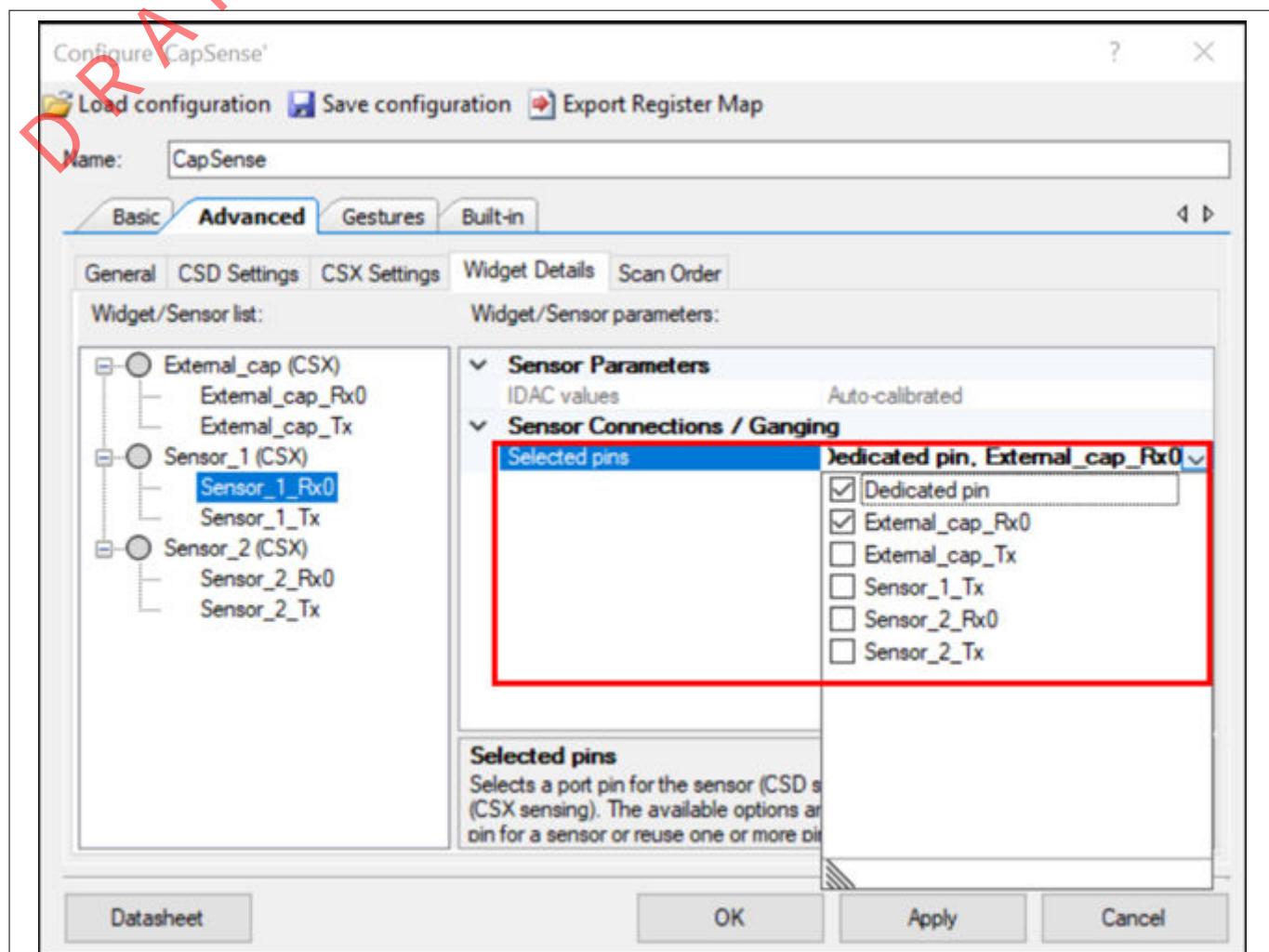


Figure 296 Gaining the external capacitor and sensor pin

5 PSoC™ 6 application notes

5.8.6 Gesture in CAPSENSE™

5.8.6.1 Touch gesture support

The CAPSENSE™ Component in PSoC™ 4 and PSoC™ 6 MCU supports the gesture detection feature for sliders and touchpad widgets. It allows to identify different predefined gestures based on touch patterns on sliders and touchpad widget.

Note: *The gesture detection feature is available for selected device part numbers. If you intend to use the gesture feature of the component, ensure that you select the device that supports this feature.*

5.8.6.2 Gesture groups

Gestures are divided into several groups: Click, one-finger scroll, two-finger scroll, two-finger zoom, one-finger edge swipe, one-finger flick, and one-finger rotate.

Table 55 lists the gestures supported by various widgets. See [Component datasheet/middleware document](#) for more details on how these gestures are defined and the parameter that to be configured in the CAPSENSE™ configurator to detect these gestures.

Table 55 Gesture supported by different CAPSENSE™ widgets

Widget type	Gesture groups						
	Click	One-finger scroll	Two-finger scroll	One-finger flick	One-finger edge swipe	Two-finger zoom	One-finger rotate
Button	✓	–	–	–	–	–	–
Linear slider	–	✓	–	✓	–	–	–
Radial slider	✓	–	–	–	–	–	–
Matrix buttons		–	–	–	–	–	–
Touchpad	✓	–	–	✓	–	–	✓
Proximity		–	–	–	–	–	–

5.8.6.3 One-finger gesture implementation

Implementing gesture detection involves following steps:

1. [Tuning the widget](#)
2. [Selecting predefined gesture](#)
3. [Firmware implementation with timestamp](#)
4. [Tuning gesture parameters](#)

5.8.6.3.1 Tuning the widget

Tune the CAPSENSE™ hardware and software parameters for the widget. Generally, in a gesture application, because of the speed and orientation of the finger movement changes, the finger may make a very little contact with the widget. This could be confirmed by viewing the centroid data in the [Tuner GUI](#) when the gesture is being performed. If the sensitivity is good enough, you will get the data without any break. If you observe any break in the centroid data, increase the sensitivity until the data for the gesture is complete and appear without any break.

5 PSoC™ 6 application notes

~~DRAFT~~

Ensure that you get a SNR above 5:1 for the slight finger contact that you may want to detect. Also, ensure that you have a linear centroid response with respect to the finger position on the slider or touchpad. Tune the sensors using guidelines in section [Slider widget tuning](#) tuning or [Touchpad widget tuning](#) for achieving the same

5.8.6.3.2 Selecting predefined gesture

First, enable Gestures in the Gesture tab in CAPSENSE™ Component. All gesture-related configuration parameters appear after enabling gestures; these parameters are systematically arranged by widgets/gesture groups as [Table 55](#) shows. According to the application requirement, you can enable and disable gestures by selecting the specific checkbox. Do the following to enable gestures and configure the corresponding parameters.

- Select the widget for which gesture feature must be enabled in the Widget pane. If you have multiple widgets in the project, the PSoC™ Creator allows gesture recognition only one widget. However, in ModusToolbox™, gesture recognition can be enabled on more than one widget
- Select the desired gestures in **Gestures** pane. User has an option to select multiple gestures. In PSoC™ Creator, you cannot enable scroll gesture and flick gesture at the same time. This is applicable for both sliders and touchpad. However, in ModusToolbox™, you can enable more than one gesture according to the application requirement
- Configure all parameters in the Parameter pane. When a gesture is selected, the right pane of the window displays the parameters of the selected gesture group. See the [Component datasheet/middleware document](#)

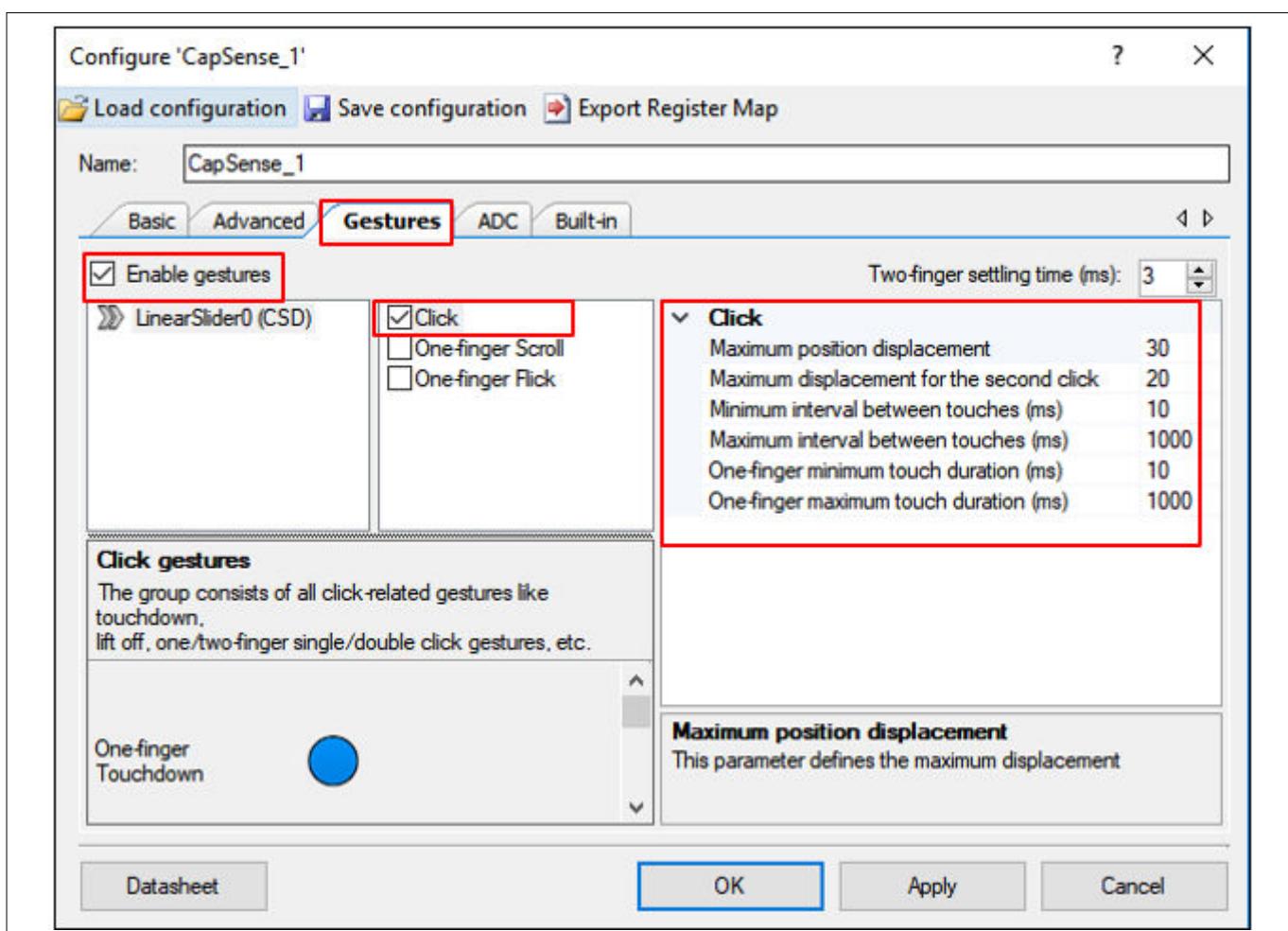


Figure 297 Configuring gestures in CAPSENSE™ component

5 PSoC™ 6 application notes

5.8.6.3.3 ~~DRAFT~~ Firmware implementation with timestamp

See the code example [PSoC™ 4 CAPSENSE™ Touchpad Gestures](#) to understand how to implement timestamp for gesture recognition. Because each gesture has a pattern of touch that changes with time, a reference timestamp is needed for properly getting the touch data with respect to time. This time stamp represents the sampling rate for the gesture recognition algorithm. Both the centroid positions and their respective timestamp are used by the gesture decoding API to determine different predefined gesture patterns that are applicable for the widget.

First, tune the widget using the procedure described in [Tuning the widget](#) and determine the time interval between two successive CAPSENSE™ scans in the firmware. Update the timestamp exactly with this duration. The way to accurately determine it is to toggle a GPIO in the firmware after the CAPSENSE™ scan is complete and find the time duration using an oscilloscope.

5.8.6.3.4 Tuning gesture parameters

This section describes how to set gesture parameters for sliders; the same procedure could be extended to the gesture groups supported by touchpads. CAPSENSE™ sliders support Click, One-finger Scroll, and One-finger flick gesture features. See the [Component datasheet/middleware document](#).

Using tuner GUI for tuning gesture parameters

You can use the **Gesture View** in [Tuner GUI](#) for tuning the gesture parameters and visualize and analyze the performance of the gesture detected in the end system.

Ensure the following while using [Tuner GUI](#) for gestures:

1. For tuning gesture parameters in runtime, [Tuner GUI](#) must be used with EZI2C. Use Synchronized communication mode for visualizing the detected gestures in runtime. For more details on using the [Tuner GUI](#), see the [Component datasheet/middleware document](#) and the [PSoC™ 4 CAPSENSE™ touchpad gestures code example](#). All the parameters for the gestures that are available in the CAPSENSE™ configurator are available in [Tuner GUI](#), where you can directly edit these values for tuning.
2. As Figure 298 shows, the Gesture View tab is organized into different panes as follows:
 - Gesture Event History** pane shows detected gestures and their positions on the widget.
 - Detected Gesture** pane indicates the detected gesture. If the delay checkbox is enabled, a gesture picture is displayed for the specified time-interval; if delay is disabled, the last reported gesture picture is displayed until a new gesture is reported.
 - Cypress® Icon** in the Tuner GUI moves according to the scroll gesture. It indicates how well the parameter of the scroll gesture is tuned. This dynamic feature gives performance feedback for further fine-tuning gesture parameters.

5 PSoC™ 6 application notes

DRAFT

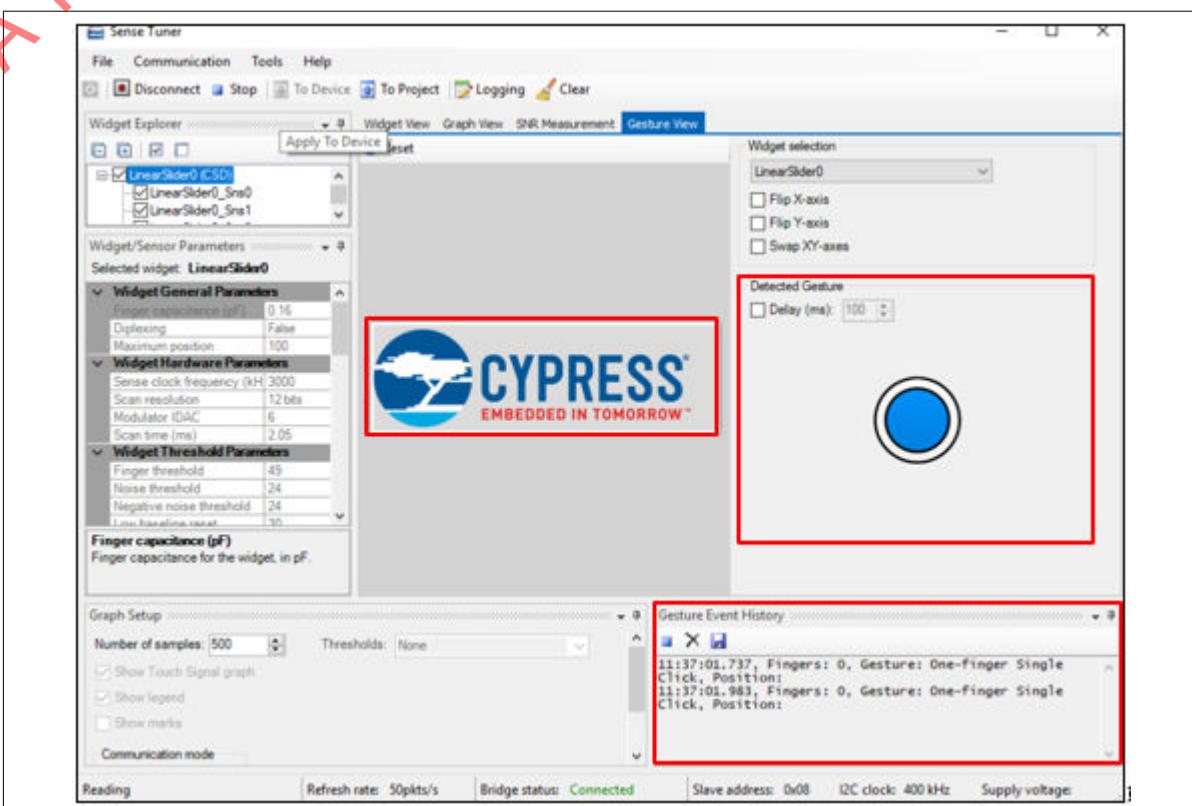


Figure 298 Tuner GUI for gestures

3. Determining the event duration using [Tuner GUI](#). A general equation to determine the event duration is given by [Equation 72](#).

$$\text{Event duration} = \text{No. of samples} \times T_{\text{sample}}$$

Equation 72 Gesture duration

Where,

No. of Samples = Number of samples the gesture event occurred. This data could be obtained from the Graph View in the [Tuner GUI](#).

T_{sample} = Time interval between two samples.

$$T_{\text{sample}} = \frac{1}{\text{Refresh rate}}$$

5 PSoC™ 6 application notes

DRAFT

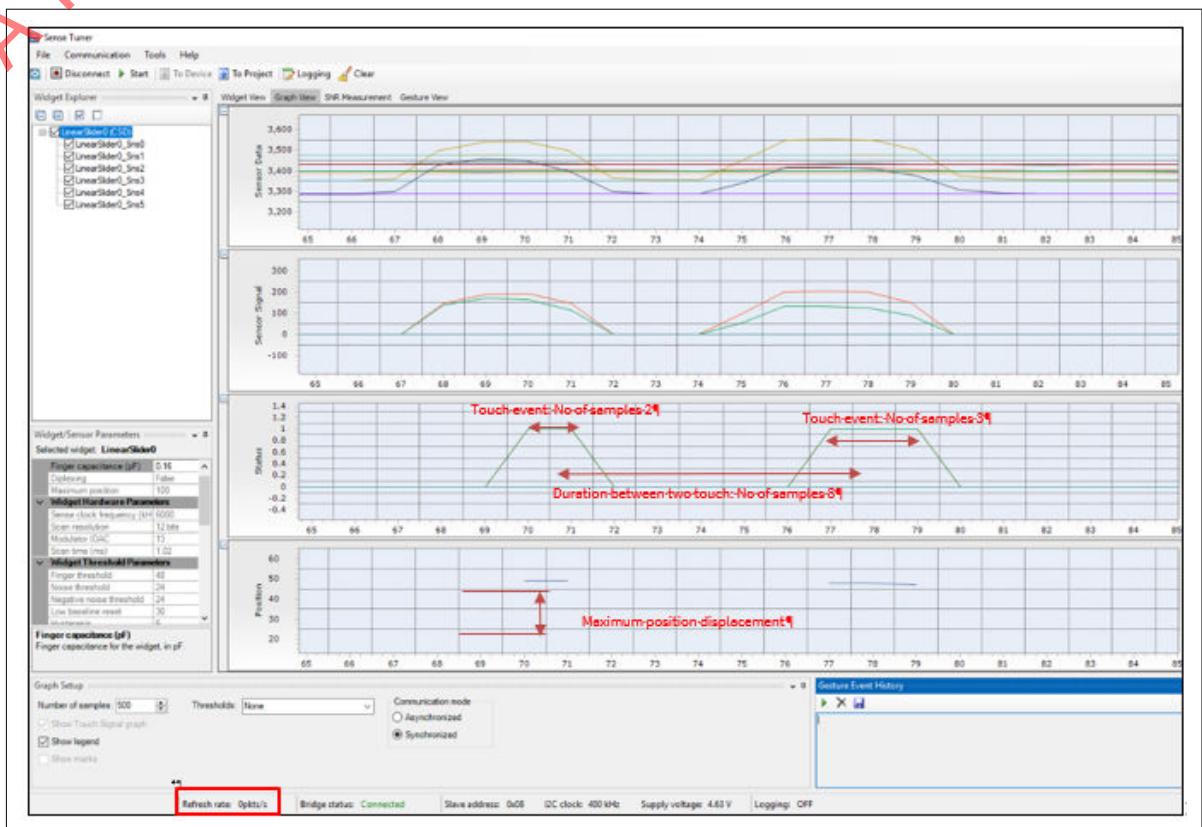


Figure 299

Determining the Gesture parameters using Tuner GUI

Click

There are two type of click gestures: single-click and double-click. [Table 56](#) lists the parameters to be configured for the Click **gesture** in both PSoC™ Creator and in ModusToolbox™. See [Component datasheet/middleware document](#). [Table 57](#) provides the recommended values of the gesture parameter for the Click gesture.

Table 56 Click gesture parameters

Gesture	PSoC™ Creator	ModusToolbox™
Single-click	One finger minimum touch duration	Minimum click timeout
	One finger maximum touch duration	Maximum click timeout
	Maximum position displacement	Maximum click distance
Double-click	Minimum interval between touches	Minimum second click interval
	Maximum interval between touches	Maximum second click interval
	Maximum displacement for the second click	Maximum second click distance

Table 57 Recommended values for click gestures

Parameters	Typical values
Maximum position displacement	20% of maximum position of the slider
Maximum position displacement for the second click	20% of maximum position of the slider
Minimum interval between touches (ms)	60

(table continues...)

~~DRAFT~~
5 PSoC™ 6 application notes

Table 57 (continued) Recommended values for click gestures

Parameters	Typical values
Maximum interval between touches (ms)	400
One finger minimum touch duration (ms)	20
One finger maximum touch duration (ms)	400

Single click

A single click is defined as a touch-down event followed by a lift-off. [Figure 300](#) shows the spatial and timing condition that must be satisfied for a valid single-click event.

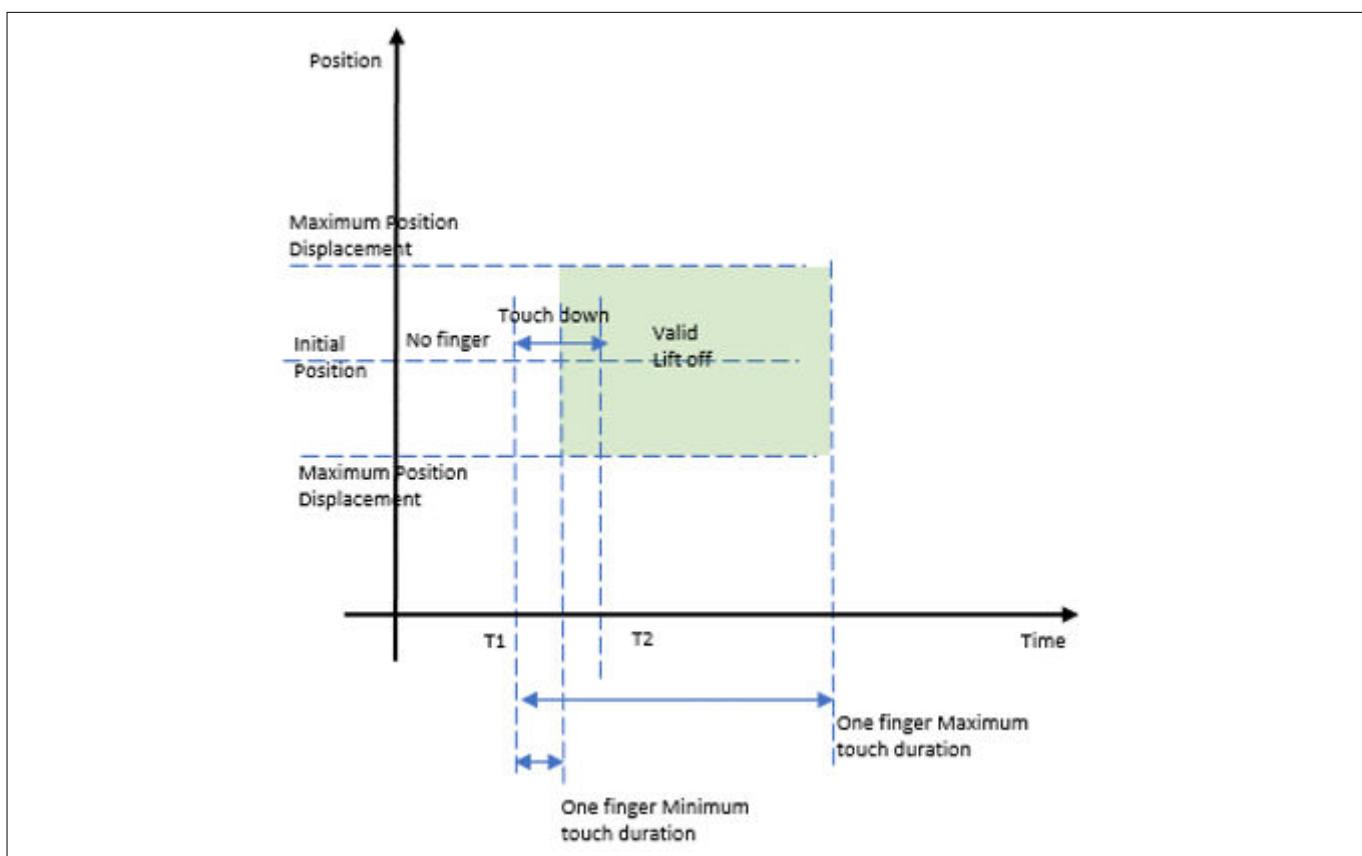


Figure 300 Single-click gesture

From [Figure 300](#), at time T1, the finger touched down on the slider; at time T2, the finger is lifted off from the slider. For a valid single click, the touch-down duration should be between the “One finger minimum touch duration” and “one-finger maximum touch duration” and the relative position of the liftoff from the initial position of touch should be less than the “Maximum position displacement” parameter.

The duration of each single-click event can be determined by using [Equation 72](#) by finding the number of samples for the single click in the **Graph** view of [Tuner GUI](#) and the refresh rate as shown in [Figure 299](#).

From the single-click event duration, fix the parameters “One-finger minimum touch duration” and “One-finger maximum touch duration”. The maximum position displacement parameter can be determined by observing the maximum variation in the centroid position using the [Tuner GUI](#) as shown in [Figure 299](#). The recommended value is 20 percent of the maximum centroid position of the slider as mentioned in [Table 56](#).

~~5 PSoC™ 6 application notes~~

~~Double click~~

A double click is two single-clicks event occurring one after another with the second click occurring between the minimum and maximum time interval between the two touches. In addition, the relative position of the second click from the initial position of touchdown event should be less than the maximum position displacement for the second click. [Figure 301](#) shows the spatial and timing conditions that must be satisfied for a valid double-click event.

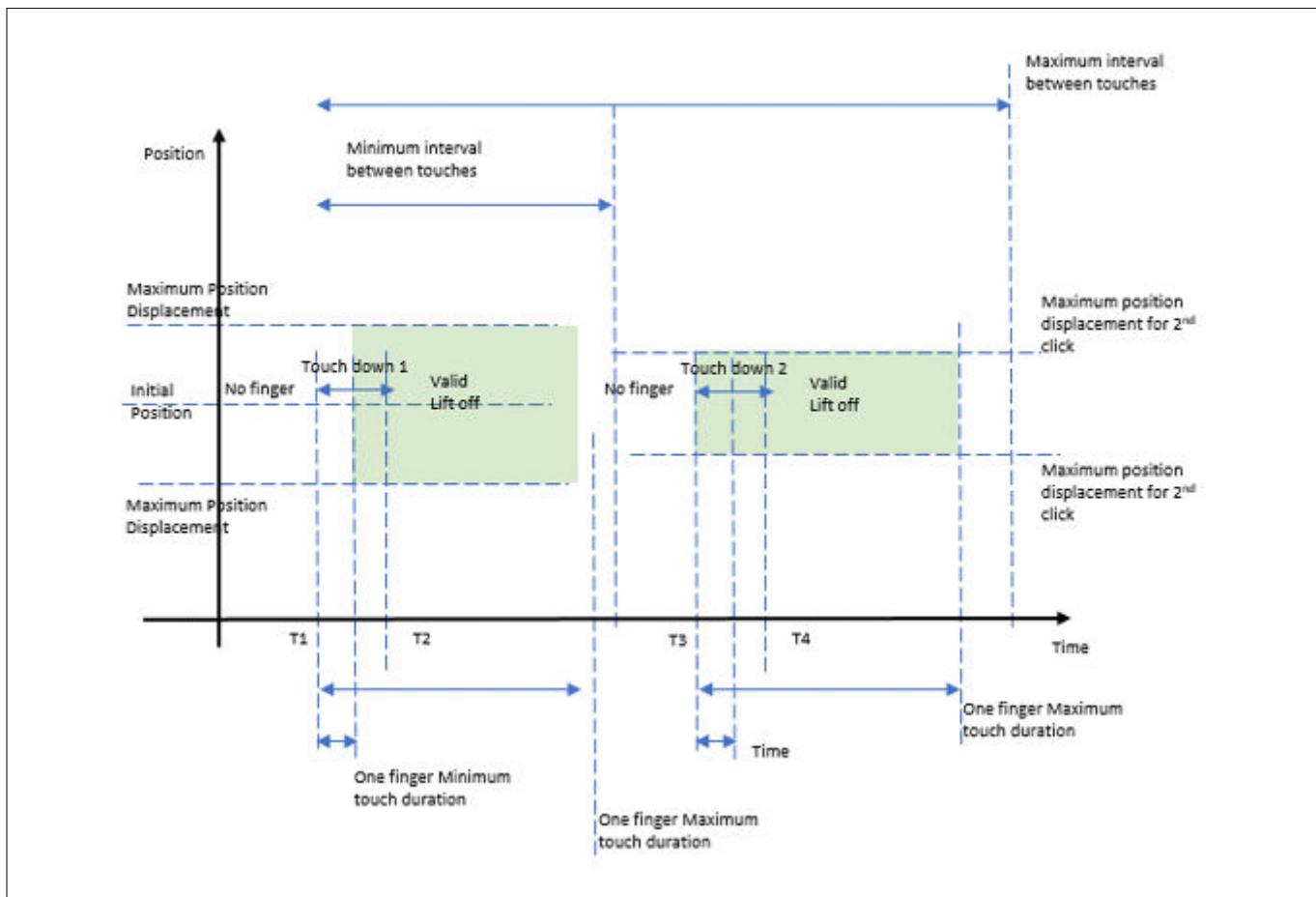


Figure 301 Double-click gesture

From [Figure 301](#), at time T1, the finger touched down on the slider for the first click; at time T2, the finger is lifted off from the slider. At T3, the finger touched down on the slider for the second click; at T4, the finger is lifted off from the slider. For a valid double click, each click should satisfy the condition of single click, and the second click should occur between Minimum and Maximum interval between touch parameters.

Using the **Graph** view in the [Tuner GUI](#), observe the double-click touch data. Determine the parameter of single click as mentioned in the [Single click](#) section. Determine the duration between the two touches using the Graph view in the [Tuner GUI](#) and set the value of the minimum and maximum intervals between touch parameters. A typical captured data for the double-click event is shown in [Figure 299](#).

Scroll

There are two different scroll gestures that can be detected on sliders: One-finger scroll and One-finger Inertial scroll. See [Component datasheet/middleware document](#). [Table 58](#) shows the parameters to be configured for the scroll gesture.

Note: One-finger Inertial Scroll gesture is not supported in ModusToolbox™.

DRAFT

5 PSoC™ 6 application notes

Table 58 One finger scroll parameters

Gesture	PSoC™ Creator	ModusToolbox™
One-finger scroll	Position threshold N	Minimum scroll distance
	Scroll step	-
	Debounce	Scroll debounce
One-finger inertial scroll	Position inertial threshold	NA
	Count level	

One-finger scroll

A One-finger Scroll gesture is a combination of a touchdown followed by a displacement in specific direction. The change in position between two consecutive scans must exceed the Position Threshold value given in the configurator after tuning. See [Component datasheet/middleware document](#).

Follow these steps to set the scroll gesture parameter values as shown in [Table 58](#).

1. Determine the number of samples of the scroll gesture from the **Graph** view (Centroid position) in [Tuner GUI](#)
2. By using [Equation 72](#) determine the duration of the complete scroll
3. Determine the change in centroid position for the complete scroll using the [Tuner GUI](#)
4. Determine Position Threshold [Equation 73](#). Each gesture is scanned at a sample rate that is set in the timestamp in the application code. The position threshold is given by the change in the centroid position for the duration that is set in the timestamp

$$\text{Position Threshold} = \frac{\text{Change in Centroid position}}{\text{Total duration of scroll}} \times \text{Duration of timestamp}$$

Equation 73 Equation to determine position threshold

5. In PSoC™ Creator, set four different position thresholds and their scroll count values in the configurator, which are determined by varying the speed of the scroll gesture. Now, change the speed of scroll and repeat the steps 1 – 4 and set these position threshold values. In ModusToolbox™ has only one parameter: Minimum Scroll distance; determine its value in the same way you determined the position threshold
6. Read the scroll step from the CAPSENSE™ data structure and use it to control the speed and smoothness of the scroll gesture. The scroll step depends on the position threshold. This scroll step is used in the application code to control the actual variable value to be changed with respect to scroll.
Note: *The scroll step parameter is not available in ModusToolbox™.*
7. Set the maximum slider position as ten times the dimension of the slider as a general rule. If you set scrollDistanceMin=10, everything below a 1-mm movement will not detect the scroll gesture. Everything above this number might detect a gesture

Observe the **Cypress®** icon in the Tuner GUI (see [Figure 298](#)) to get a feedback on how well the tuning has been done for the scroll gesture in the given hardware. You can also print the variable that must be controlled by scroll through UART to visualize how the value is changing with respect to scroll. This could be used as a visual feedback. The position threshold parameters and the corresponding step counts should be tuned until the variation in the variable value with respect to scroll meet the requirement of the end user application.

~~5 PSoC™ 6 application notes~~

~~One-finger inertial scroll~~

The one-finger Inertial scroll gesture is defined as a touchdown event followed by a minimum displacement in a specific direction, and then a liftoff. The movement of scroll will automatically stop when it reaches the end value of the variable. See [Component datasheet/middleware document](#).

The gesture parameter is provided in [Table 58](#). The position inertial threshold parameter is given by the minimum change in centroid position that is required before a liftoff; its value can also be determined by steps in the [One-finger scroll](#) section. The count value parameter defines the momentum of scroll; it can take two possible values: low or high. Choose the count value according to the end application requirement.

One-finger flick

A flick gesture is a touchdown event followed by a high-speed displacement and a liftoff event (see [Component datasheet/middleware document](#)). The flick gesture is similar to the One-finger Inertial Scroll; the only difference is that it requires a high-speed displacement followed by a liftoff event within the maximum sample interval defined in the configurator. You can determine the position threshold and the maximum sample interval by following the same procedure in [One-finger scroll](#) section and by using [Equation 72](#).

Table 59 One-finger flick gesture parameters

Gesture	PSoC™ Creator	ModusToolbox™
One Finger Flick Gesture	Position threshold	Minimum flick distance
	Maximum sample interval	Maximum flick timeout

5.8.6.3.5 Two-finger gesture implementation

Two-finger gestures such as Two-finger Scroll and Two-finger Zoom are supported in the touchpad widget. You must enable this feature in the **Widget Details** tab of the Touchpad Widget. The procedure for tuning the parameters is the same as mentioned in the [One-finger gesture implementation](#) section (see [Component datasheet/middleware document](#)). [Figure 302](#) shows how to enable two-finger touch gestures in the configurator, select the centroid type as **5 x 5 Centroid**, and set the two-finger detection as **True**.

5 PSoC™ 6 application notes

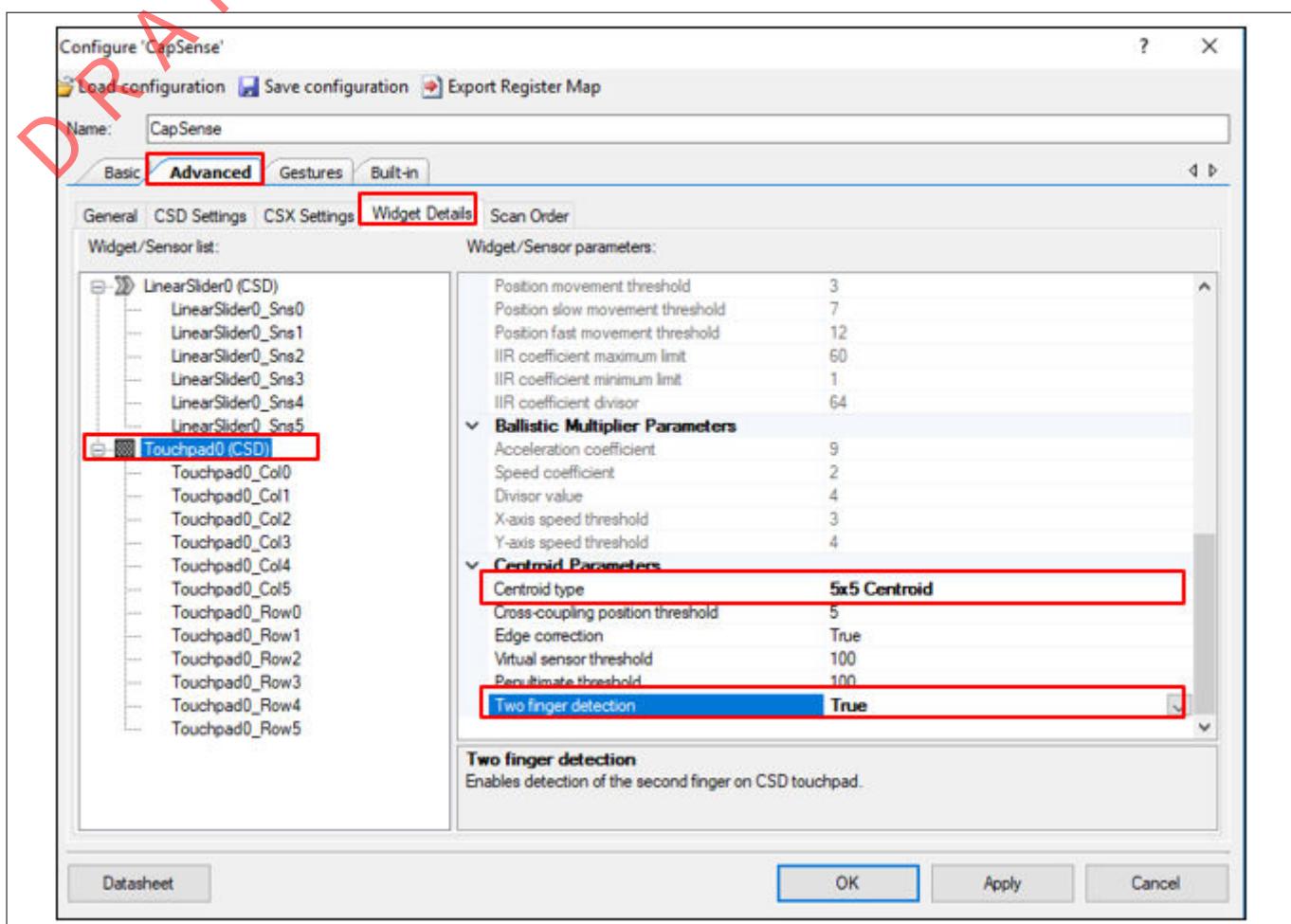


Figure 302 Enabling two-finger touch gestures in the CAPSENSE™ component

5.8.6.3.6 Advanced filters for gestures

Advanced filtering features for gestures such as Ballistic multiplier, Adaptive IIR filter, and the Edge correction feature are available to improve gesture recognition and the user experience.

See [Component datasheet/middleware document](#).

5 PSoC™ 6 application notes**5.8.7 Design considerations**

This chapter explains the firmware and hardware design considerations for CAPSENSE™.

5.8.7.1 Firmware

The CAPSENSE™ component provides multiple application programming interfaces to simplify firmware development. The CAPSENSE™ Component datasheet provides a detailed list and explanation of the available APIs. You can use the CAPSENSE™ Example projects provided in PSoC™ Creator or ModusToolbox™ to learn schematic entry and firmware development. See Chapter 5.8.4 for more details.

The CAPSENSE™ scan is non-blocking in nature. The CPU intervention is not required between the start and the end of a CAPSENSE™ scan. Therefore, you can use CPU to perform other tasks while a CAPSENSE™ scan is in progress. However, note that CAPSENSE™ is a high-sensitive analog system. Therefore, sudden changes in the device current may increase the noise present in the raw counts. If you are using widgets that require high

5 PSoC™ 6 application notes

sensitivity such as proximity sensors, or buttons with thick overlay, you should use a blocking scan. Example firmware for a non-blocking scan is shown below.

```
/* Enable global interrupts */
CyGlobalIntEnable;

/* Start EZI2C component */
EZI2C_Start();

/*
 * Set up communication data buffer to CapSense data structure to be
 * exposed to I2C master at primary slave address request.
*/
EZI2C_EzI2CSetBuffer1(sizeof(CapSense_dsRam),
sizeof(CapSense_dsRam),
(uint8 *)&CapSense_dsRam);

/* Initialize CapSense component */
CapSense_Start();
/* Scan all widgets */
CapSense_ScanAllWidgets();

for(;;)
{
    /* Do this only when a scan is done */
    if(CapSense_NOT_BUSY == CapSense_IsBusy())
        { /* Process all widgets */
            CapSense_ProcessAllWidgets();
            /* Scan result verification */
            if (CapSense_IsAnyWidgetActive())
                {
                    /* Add any required functionality
                     based on scanning result */
                }
            /* Include Tuner */
            CapSense_RunTuner();
            /* Start next scan */
            CapSense_ScanAllWidgets();
        }
    /* CPU Sleep */
    CySysPmSleep();
}
```

You should avoid interrupted code, power mode transitions, and switching ON/OFF peripherals while a high-sensitivity CAPSENSE™ scan is in progress. However, if you are not using high-sensitivity widgets, you can use CPU to perform other tasks. You can also use low-power mode of PSoC™ to reduce the average power consumption of the CAPSENSE™ system, as explained in the next section. Monitoring and verifying the raw counts and SNR using the Tuner GUI is recommended if you are using a non-blocking code.

If you want to develop firmware using the ModusToolbox™ software, see the references in the [ModusToolbox™](#) section of this document.

5 PSoC™ 6 application notes~~DETAILED DESIGN~~
5.8.7.1.1 Low-power design

PSoC™ low-power modes allow you to reduce overall power consumption while retaining essential functionality. See [AN86233 - PSoC™ 4 low-power modes and power reduction techniques](#), for a basic knowledge of PSoC™ 4 low-power modes, see [AN219528 - PSoC™ 6 MCU low-power modes and power reduction techniques](#), for PSoC™ 6's low-power modes and [AN210998 - PSoC™ 4 low-power CAPSENSE™ design](#), for design a low-power CAPSENSE™ application.

The CPU intervention is not required between the start and the end of a CAPSENSE™ scan. If the firmware does not have any additional task other than waiting for the scan to finish, you can put the device to Sleep mode after initiating a scan to save power. When the CSD hardware completes the scan, it generates an interrupt to return the device to the Active mode.

There are different firmware and hardware techniques to reduce the power consumption of the CAPSENSE™ system.

1. If you use APIs that scan multiple widgets together, the device returns to Active mode after finishing the scan of a single widget. Therefore, you should scan each widget individually for reducing the power consumption in the design. See the CAPSENSE™ Component datasheet
2. You can use the Deep-Sleep mode of PSoC™ to considerably reduce the power consumption of a CAPSENSE™ design. However, the CAPSENSE™ hardware is disabled in the Deep-Sleep mode. Therefore, the device must wake up frequently to scan for touches. You can use the watchdog timer (WDT) in PSoC™ to wake up the device from the Deep-Sleep mode at frequent intervals. Increasing the frequency of the scans improves the response of the CAPSENSE™ system, but it also increases the average power consumption
3. As the number of sensors in the design increases, the device has to spend more time in the Active mode to scan all sensors. This, in turn, increases the average power consumption. For saving power in a design with multiple sensors, you should include a separate [proximity loop](#) that surrounds all the sensor. When the device wakes up from the Deep-Sleep mode, only scan this proximity sensor. If the proximity sensor is active, the device must stay in the Active mode and scan other sensors. If the proximity sensor is inactive, the device can return to the Deep-Sleep mode. [Figure 303](#) illustrates this process

5 PSoC™ 6 application notes

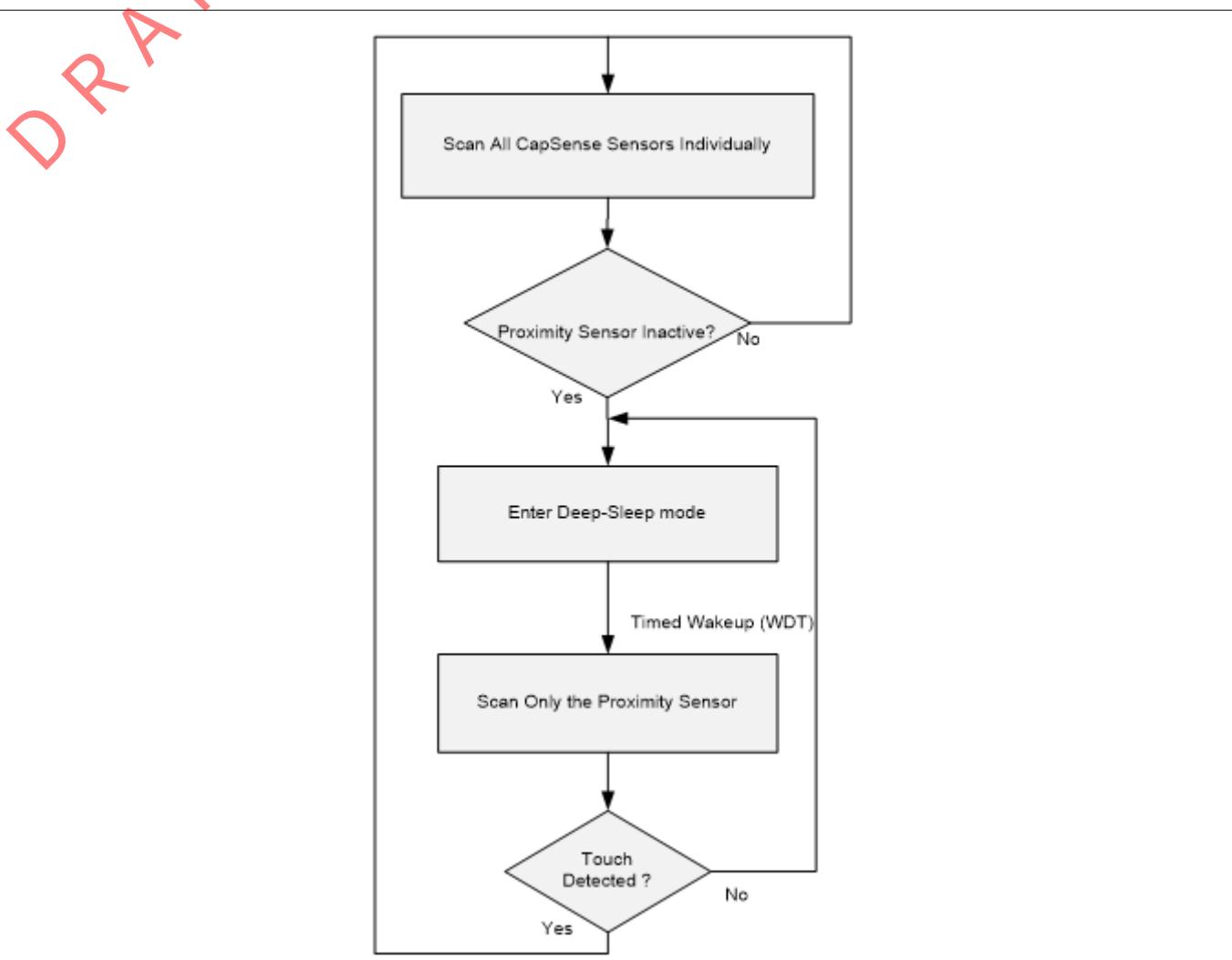


Figure 303 Low-power CAPSENSE™ Design

1. The CAPSENSE™ component can reduce power consumption by reducing the execution time of scanning by ganging sensors together and managing scanning at the application level. In this case, all the sensors in the design are “ganged” that is, simultaneously connected to the AMUX bus to form a virtual sensor. See the code example [PSoC™ 4 low power ganged sensor](#) and [AN92239 - Proximity sensing with CAPSENSE™](#) for details on ganged sensor implementation. A ganged sensor has different tuning parameters because its properties are different compared to considering the sensors individually. Therefore, it should be considered as a single CSD button and tuned separately; see [Manual tuning](#). The ganged sensor is periodically scanned by using a watchdog timer (WDT); if the ganged sensor reports a touch event, enable the scanning of the actual widgets that need to be scanned. This is helpful in CAPSENSE™ designs that requires Wake on Touch modes. The procedure is similar to what is explained in [Figure 303](#). You can achieve very low system current while maintaining a good touch response, by properly tuning CAPSENSE™ and the wakeup interval. This technique could also be used with the CSX touchpad widget.
2. If high-speed peripherals such as system timers and I²C are required, you can put the CPU to sleep mode instead of going to deep sleep mode
3. You can also add a shield hatch in the design, as explained in [Driven shield signal and shield electrode](#) to reduce the parasitic capacitance and therefore, the scan time. The scan time and power consumption is directly related; thus, the power consumption is reduced by lowering the scan time

Note: In PSoC™ 4000 devices, it is not recommended to enter Sleep mode if a CAPSENSE™ scan is in progress.

~~DEAF~~ 5 PSoC™ 6 application notes

5.8.7.2 Sensor construction

A capacitive sensor can be constructed using different materials depending on the application requirement. In a typical sensor construction, a conductive pad, or surface that senses a touch is connected to the pin of the PSoC™ using a conductive trace or link. This whole arrangement is placed below a non-conductive overlay material and the user interacts on top of the overlay.

Figure 304 shows the most common CAPSENSE™ sensor construction.

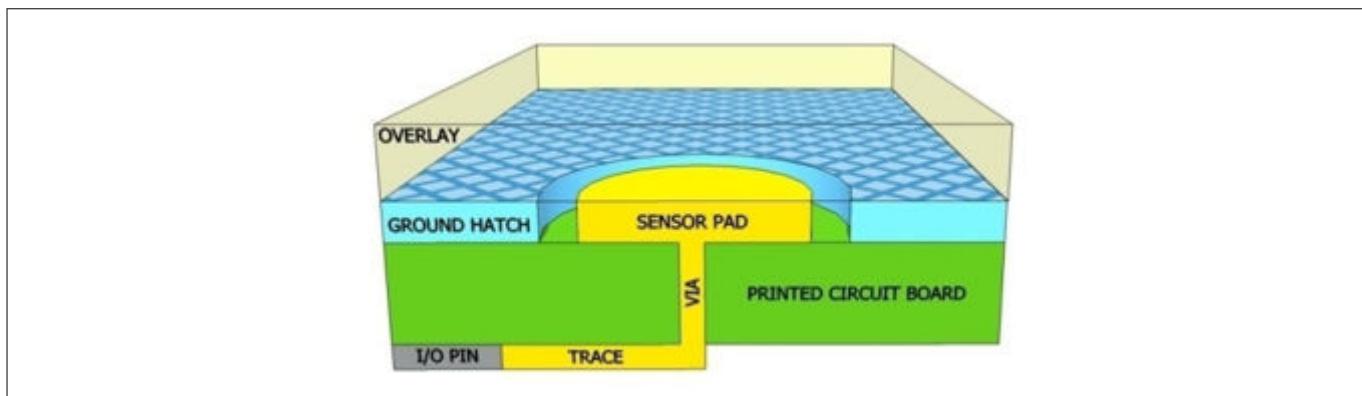


Figure 304 CAPSENSE™ sensor construction

The copper pads etched on the surface of the PCB act as CAPSENSE™ sensors. A nonconductive overlay serves as the touch surface. The overlay also protects the sensor from the environment and prevents direct finger contact. A ground hatch surrounding the sensor pad isolates the sensor from other sensors and PCB traces.

If liquid tolerance is required, you should use a shield hatch instead of the ground hatch. In this case, drive the hatch with a shield signal instead of connecting it to ground. See [Liquid tolerance](#) section for details.

The simplest CAPSENSE™ PCB design is a two-layer board with sensor pads and hatched ground plane on the top, and the electrical components on the bottom. Figure 305 shows an exploded view of the CAPSENSE™ hardware.

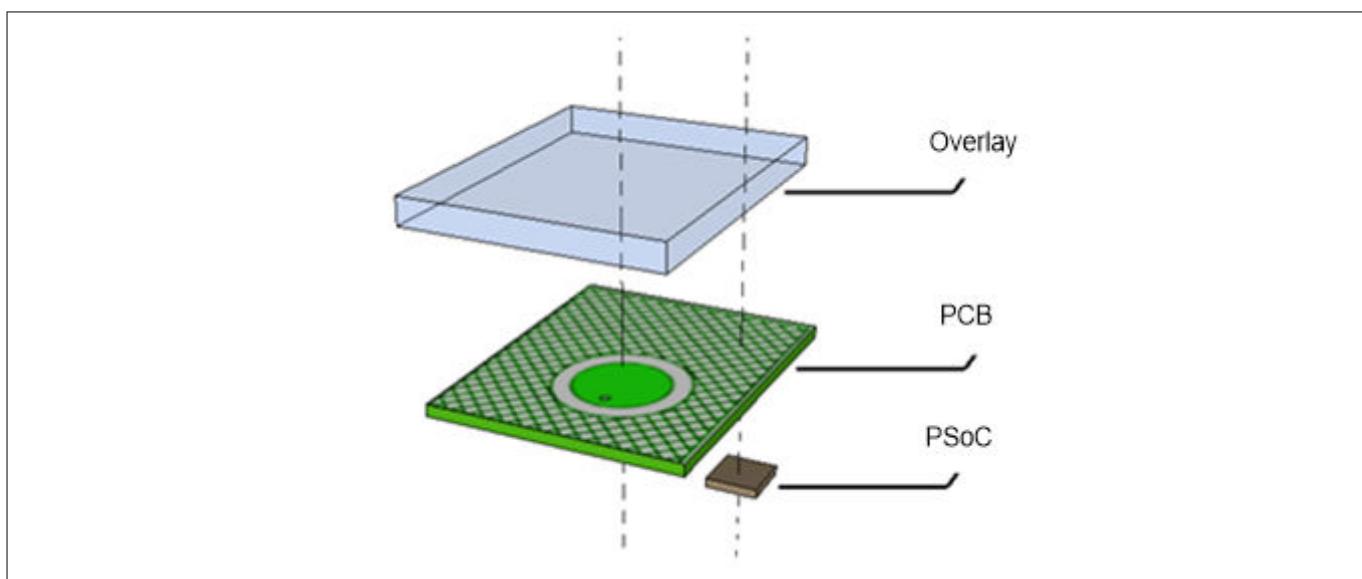


Figure 305 CAPSENSE™ hardware Sensor construction using springs as sensors

Sensors may also be constructed by using materials other than copper, such as indium tin oxide (ITO) or printed ink on substrates such as glass or a flex PCB. In some cases, springs can also be used as CAPSENSE™ sensors as Figure 306 shows, to create elevated sensors that allow overlay to be placed at an elevated distance from the

5 PSoC™ 6 application notes

PCB. See [Getting started with CAPSENSE™ design guide](#) for PCB design considerations specific to spring sensors and other non-copper sensors such as ITO and printed ink.

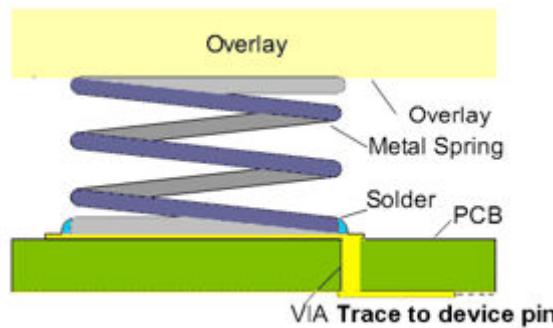


Figure 306 Sensor construction using springs as sensors

5.8.7.3 Overlay selection

5.8.7.3.1 Overlay material

The overlay is an important part of CAPSENSE™ hardware as it determines the magnitude of finger capacitance. The finger capacitance is directly proportional to the relative permittivity of the overlay material. See [Finger capacitance](#) for details.

Table 60 shows the relative permittivity of some common overlay materials. Materials with relative permittivity between 2.0 and 8.0 are well suited for CAPSENSE™ overlay.

Table 60 Relative permittivity of overlay materials

Material	ϵ_r
Air	1.0
Formica	4.6 – 4.9
Glass (Standard)	7.6 – 8.0
Glass (Ceramic)	6.0
PET film (Mylar®)	3.2
Polycarbonate (Lexan®)	2.9 – 3.0
Acrylic (Plexiglas®)	2.8
ABS	2.4 – 4.1
Wood table and desktop	1.2 – 2.5
Gypsum (Drywall)	2.5 – 6.0

Note: *Conductive materials interfere with the electric field pattern. Therefore, you should not use conductive materials for overlay. You should also avoid using conductive paints on the overlay.*

5.8.7.3.2 Overlay thickness

Finger capacitance is inversely proportional to the overlay thickness. Therefore, a thin overlay gives more signal than a thick overlay. See [Finger capacitance](#) for details.

~~5 PSoC™ 6 application notes~~

~~DRAFT~~ Table 61 lists the recommended maximum thickness of acrylic overlay for different CAPSENSE™ widgets.

Table 61 Maximum thickness of acrylic overlay

Widget	Maximum thickness (mm) – 4th Generation CAPSENSE	Maximum thickness (mm) – 5th Generation CAPSENSE
Button	5	18
Slider	5 ²¹	18
Touchpad	0.5	3

Because [Finger capacitance](#) also depends on the dielectric constant of the overlay, the dielectric constant also plays a role in the guideline for the maximum thickness of the overlay. Common glass has a dielectric constant of approximately $\epsilon_r = 8$, while acrylic has approximately $\epsilon_r = 2.5$. The ratio of $\epsilon_r/2.5$ is an estimate of the overlay thickness relative to plastic for the same level of sensitivity. Using this rule of thumb, a common glass overlay can be about three times as thick as a plastic overlay while maintaining the same level of sensitivity.

In addition, avoid using very thin or no overlay. It is important to have a minimum overlay thickness in a CAPSENSE™ design for the following reasons:

1. An overlay provides protection from the environmental condition, prevents direct finger contact, and gives ESD protection. The overlay thickness should be small enough to give a good signal, and decided based on the button size and the strength to withstand ESD. See [AN64846 - Getting started with the CAPSENSE™](#)
2. For the CSD button, if there is no overlay the buttons will be over sensitive
3. For sliders, if there is no overlay, the raw count may saturate for the slider segments and may cause non-linear centroid response for slider. See [Slider design](#)
4. For the CSX sensor, it is recommended to have a minimum overlay thickness of 0.5 mm. If it is violated, sudden decrease in raw count is observed when a finger is placed on a sensor or a water drop falls on the Tx and Rx electrodes. See [Effect of grounding](#)

5.8.7.3.3 Overlay adhesives

The overlay must have a good mechanical contact with the PCB. You should use a nonconductive adhesive film for bonding the overlay and the PCB. This film increases the sensitivity of the system by eliminating the air gap between the overlay and the sensor pads. 3M™ makes a high-performance acrylic adhesive called 200 MP that is widely used in CAPSENSE™ applications. It is available in the form of adhesive transfer tapes; example product numbers are 467 MP and 468 MP.

5.8.7.4 PCB layout guidelines

PCB layout guidelines help you to design a CAPSENSE™ system with good sensitivity and high [Signal-to-noise ratio \(SNR\)](#).

5.8.7.4.1 Sensor CP

In a CAPSENSE™ system design, the C_P of the sensor must be within the supported range of the device. You can find the supported C_P range in the [Component datasheet/middleware document](#). The main components of C_P are trace capacitance, sensor pad capacitance, and pin capacitance of the device. The pin capacitance is device-dependent (see the [Device datasheet](#)), so you can only design your sensor and trace capacitance to be able to meet the C_P criteria in the datasheet. The relationship between C_P and the PCB layout features is not

²¹ For a 5 mm acrylic overlay, the SmartSense Component requires a minimum of 9 mm finger diameter for slider operation. If the finger diameter is less than 9 mm, Manual Tuning should be used.

5 PSoC™ 6 application notes

simple. C_p increases with an increase in the sensor pad size and trace length and width, and with a decrease in the gap between the sensor pad and the ground hatch.

There are many ways to decrease the C_p :

- Decrease the trace length and width as much as possible. Reducing the trace length increases noise immunity
- Drive the hatch with a shield signal. See [Driven shield signal and shield electrode](#)

Reducing the sensor pad size is not recommended because it also reduces the finger capacitance. In some special cases, such as small sensor pad and very small trace length due to placement of the sensor pad close to the device, there is a possibility of the sensor C_p to be lower than the supported minimum C_p by the device. In that case, add a footprint of the capacitor across the sensor or any unused pin. If the C_p is identified to be lower than the supported range, place a 4.7 pF capacitor across the sensor or on the unused pin and gang the capacitor during the CAPSENSE™ scan, refer to the FAQ [I am unable to calibrate my system to 85 percent](#) for more details. This will increase the C_p of the sensor to the supported range.

If the sensor C_p is very high due to long traces or because of a nearby ground, use the mutual-capacitance sensing method so that the sensitivity is not degraded because of the high C_p value. The sensitivity of the CAPSENSE™ sensor in a mutual-capacitance sensing method is independent of the sensor C_p .

5.8.7.4.2 Board layers

Most applications use a two-layer board with the sensor pads and the hatched ground planes on the top side and all other components on the bottom side. PCBs that are more complex use four layers.

- FR4-based PCB designs perform well with board thickness ranging from 0.020 inches (0.5 mm) to 0.063 inches (1.6 mm).
- Flex circuits work well with CAPSENSE™ too. You can use flex circuits for curved surfaces. All PCB guidelines in this document also apply to flex. You should use flex circuits with thickness 0.01 inches (0.25 mm) or higher for CAPSENSE™. The high breakdown voltage of the Kapton® material (290 kV/mm) used in flex circuits provides built in ESD protection for the CAPSENSE™ sensors

5.8.7.4.3 Button design

Self-capacitance button design

The self-capacitance button has a single electrode and can have different shapes and size as recommended below.

Shape: You should use circular sensor pads for CAPSENSE™ buttons. Rectangular shapes with rounded corners are also acceptable. However, you should avoid sharp corners (<90°) since they concentrate electric fields.

[Figure 307](#) shows recommended button shapes.

5 PSoC™ 6 application notes

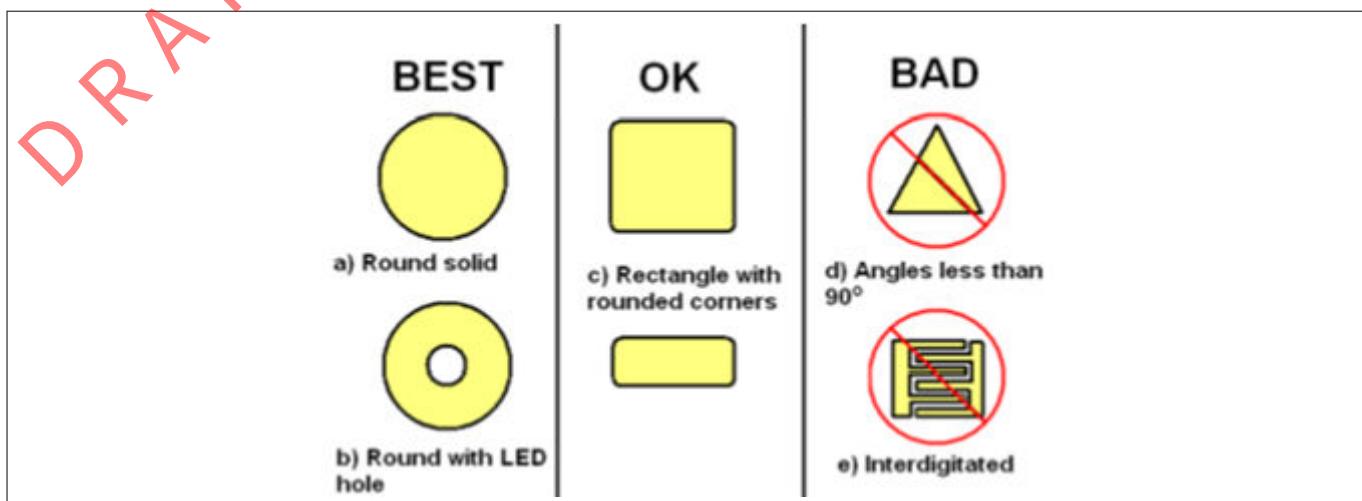


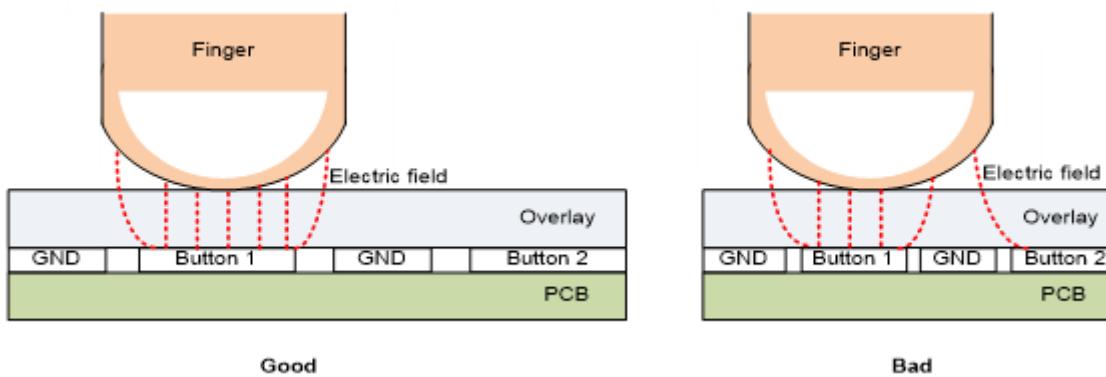
Figure 307 Recommended button shapes

Size: Button diameter should be 5 mm to 15 mm, with 10 mm suitable for most applications. A larger diameter is appropriate for thicker overlays.

Spacing: The width of the gap between the sensor pad and the ground hatch should be equal to the overlay thickness, and range from 0.5 mm to 2 mm. For example, if the overlay thickness is 1 mm, you should use a 1 mm gap. However, for a 3 mm overlay, you should use a 2 mm gap.

Select the spacing between two adjacent buttons such that when touching a button, the finger is not near the gap between the other button and the ground hatch, to prevent false touch detection on the adjacent buttons, as [Figure 308](#) shows.

Figure 308 Spacing between buttons



Mutual-capacitance button design

Mutual capacitance sensing measures the change in capacitive coupling between two electrodes. The sensor pattern should be designed in such a way that the finger disturbs the electric field between the Tx and Rx electrodes to a maximum extent. There are standard button patterns that could be used for the mutual capacitance button design and their parameters could be modified based on the application requirement. Fishbone pattern is one of the mutual capacitance patterns which give better performance in terms of SNR.

Fishbone pattern

Prongs or fishbone are standard shapes for mutual-capacitance buttons. The Tx forms a box or ring around the button for shielding Rx from noise. There are interlaced Tx and Rx prongs inside the border to form the electric

5 PSoC™ 6 application notes

~~DRAFT~~

field. Figure 309 shows an example of a two-prong fishbone sensor structure with top and bottom view with hatched ground. The gap between the outer wall of the Tx electrode and the coplanar hatch ground should be greater than the air-gap of Tx and Rx electrodes. The reference plane (PCB bottom layer) of the Fishbone structure should have void region as shown in Figure 309.

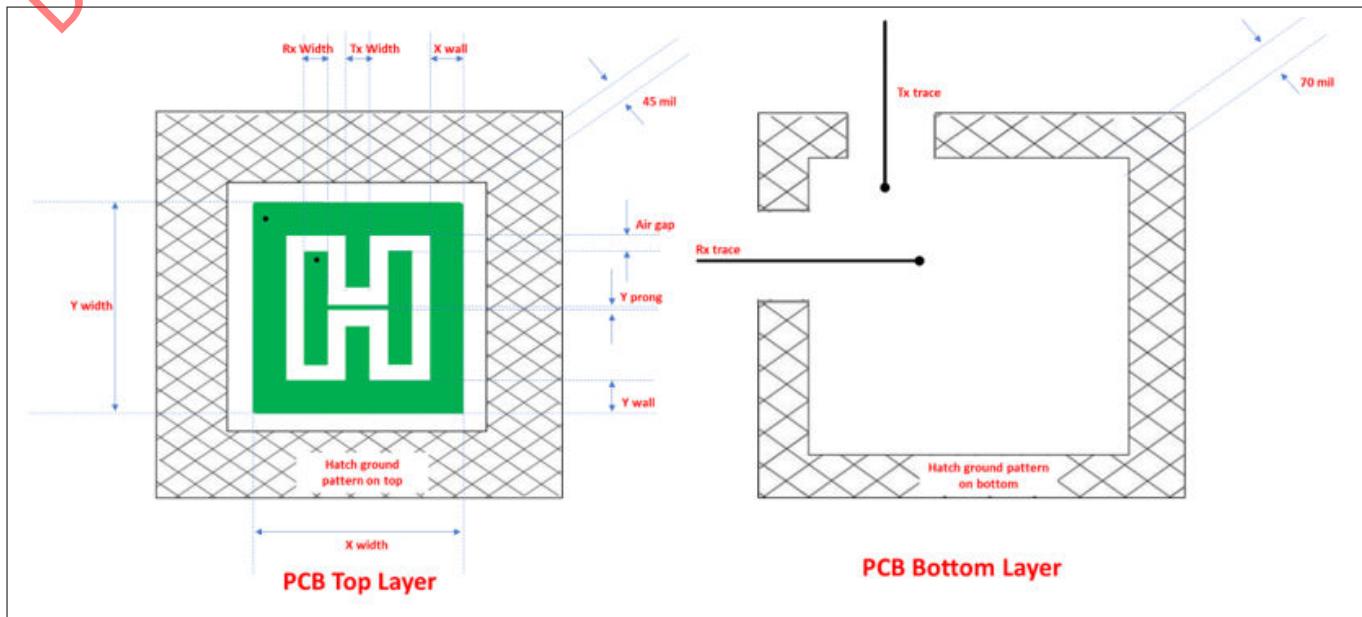


Figure 309 CSX Fish bone button pattern with two Rx prongs

Table 62 lists the suggested fishbone button design parameters for some commonly used sensor sizes and overlay like glass and acrylic. As explained in section [Sensor size](#), the recommended button size is to keep the button X and Y dimension close to the sum of expected user finger size and overlay thickness. However, the table lists multiple button sizes that you can chose from if you have constraints on available space on board or if you would like to have a bigger button for your application for ease of user interaction etc.

Also, note that for a given button size, the achievable SNR decreases with increased overlay thickness. Thus, if you plan to use thick overlays (approx. > 1 mm acrylic or 2 mm glass) ensure to avoid compromising on button size due to board space because that will further limit the button SNR performance. Ensure to use bigger buttons (>= biggest expected finger size) for such thick overlays. And also, for small buttons better to have thin overlays for getting good SNR.

Table 62 Dimension of Fishbone buttons (all units in mm)

Button size (X-size, Y-size) (mm)	Number of Rx-prongs	Air gap between Tx and Rx in mm	Tx width in mm	Rx width in mm	X-wall wdth in mm	Y-wall width in mm	Y prong in mm
5, 5	3	0.35	0.48	0.48	0.24	0.24	0.2
10,7	3	0.75	0.92	0.92	0.46	0.46	0.2
10,5	3	0.5	1.17	1.17	0.58	0.58	0.2
10,10	2	0.9	1.60	1.60	0.80	0.80	0.2
12, 12	2	1.3	1.70	1.70	0.85	0.85	0.2
13, 10	2	1.1	2.15	2.15	1.08	1.08	0.2
13, 13	2	1.5	1.75	1.75	0.88	0.88	0.2
15, 15	2	1.7	2.05	2.05	1.03	1.03	0.2

(table continues...)

5 PSoC™ 6 application notes

Table 62 (continued) Dimension of Fishbone buttons (all units in mm)

Button size (X-size, Y-size) (mm)	Number of Rx-prongs	Air gap between Tx and Rx in mm	Tx width in mm	Rx width in mm	X-wall wdth in mm	Y-wall width in mm	Y prong in mm
17, 17	2	2.3	1.95	1.95	0.98	0.98	0.2
20, 13	2	1.8	3.20	3.20	1.60	1.60	0.2
25, 13	2	2	4.25	4.25	2.13	2.13	0.2

The above button design parameters in [Table 62](#) ensure a good SNR performance if you follow the schematics and layout guidelines in this chapter.

Note: *In case if you expect a higher external noise in the design and for other complex cases you can contact Technical support for any assistance in the button design. Refer to the section [Noise in CAPSENSE™ system](#) for more details about the external noise. And in the design if you expect a low C_M then follow the guidelines mentioned in the section [I am observing a low CM for my CSX button for mitigating it.](#)*

Button design for arbitrary shapes and dimensions

[Figure 310](#) shows the different orientation of Rx prongs in the Fish bone pattern, in Button A the Rx prong is perpendicular to the side of the button with larger dimension and in Button B the Rx prongs is parallel to the side of the button with larger dimension. Orientation of Rx prongs like in Button A results in optimized button pattern compared to Button B. Thus, it is always recommended to have Rx prongs perpendicular to the side of the button with larger dimension. Thus if you need a 10x13 mm button, then simply use the 13x10 mm button from [Table 61](#) and rotate it 90° to get 13x10 mm [Noise in CAPSENSE™ system](#) button pattern as shown in [Figure 310](#).

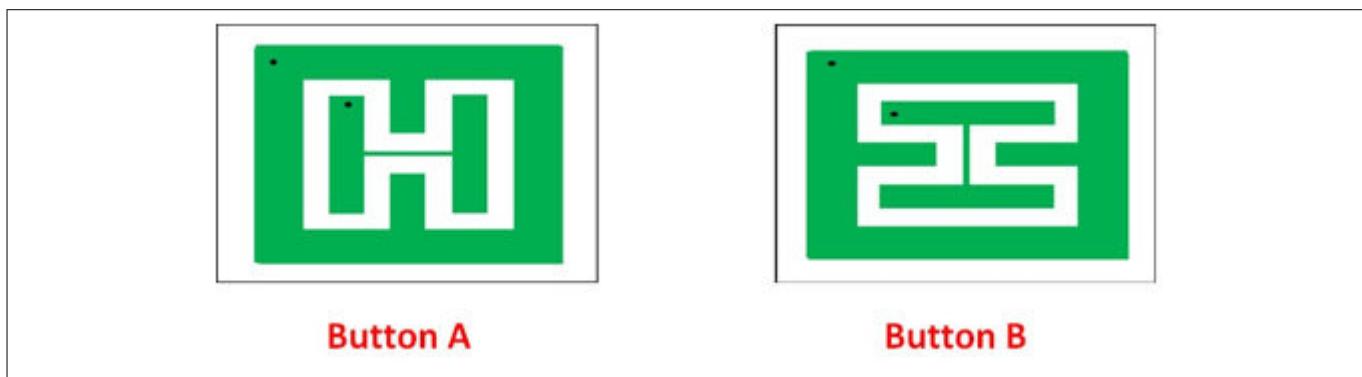


Figure 310

Orientation of Rx prongs

5 PSoC™ 6 application notes

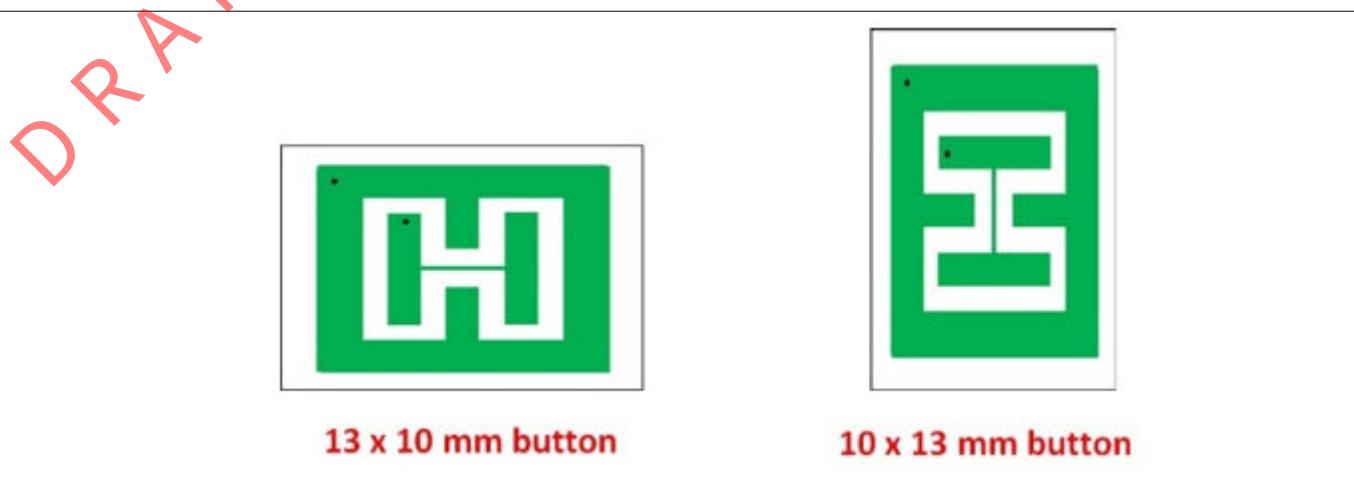


Figure 311 Rotating the button 90 deg to get the desired button dimension

There may be some design where you need a different button shape than the recommended rectangular, like an oval or circular shape etc. The below steps explain how to construct the button with nonstandard shape from the standard Fish bone pattern.

- First select the Fishbone pattern (Rectangular shape for oval shape and Square button for circular shape) from [Table 61](#) to cover the desired button shape
- Then in the user interface or above the overlay print button shape with required dimension over the fishbone pattern as shown in [Figure 312](#)

Mutual capacitance buttons designed using this method have some oversensitive area or less sensitive outside the button shape as shown in the below figures, this could be mitigated by properly tuning the software thresholds of the mutual capacitance button. The below figure shows an example of a circular button made using a square fishbone pattern.

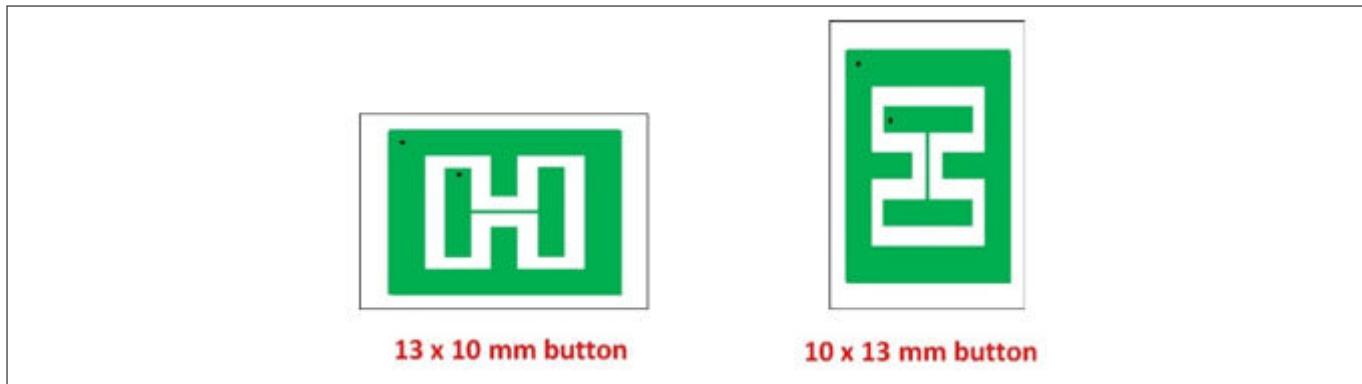


Figure 312 Arbitrary shape button design based on arbitrary pattern

If you want a pattern that is not present in [Table 61](#), you can obtain the button parameters by following few steps. For example, if you want a 19x19 pattern, choose the pattern that is close to the required pattern from [Table 61](#) like 17x17, and scale the air gap between Tx and Rx with respect to the button areas. For example:

$$New_{gap} = TxRx_{gap} \times \left(\frac{ButtonArea_{desired}}{ButtonArea_{reference}} \right)$$

Where, ButtonArea=X x Y dimension of the sensor.

Compute the Tx, Rx width and Tx wall based on the assumption below and by considering the New_{gap} as obtained above. The obtained values of the button design parameters are shown in [Table 62](#). Refer to the [Figure 309](#) to understand the description of the button design parameters.

$$Tx_{width} = Rx_{width}$$

5 PSoC™ 6 application notes

~~DRAFT~~

$$Tx_{wall} = \frac{Tx_{width}}{2}$$

Table 63 Button parameter for 19 X 19 button form 17 X 17 button

Button size (X-size, Y-size) (mm)	Number of Rx-prongs	Air gap between Tx and Rx in mm	Tx width in mm	Rx width in mm	X-wall width in mm	Y-wall width in mm	Y prong in mm
17, 17	2	2.3	1.95	1.95	0.98	0.98	0.2
19,19	2	2.9	1.85	1.85	0.93	0.93	0.2

General recommendations on Fishbone pattern parameters

Sensor size

The sensor size is the XY dimension of the button, it is selected based on the board space availability, expected user finger size and overlay material and thickness. Sensor size selection also depends upon the number of required buttons on the PCB considering required button-to-button gap and space availability in the PCB. But if the space is not the constrain then choose higher button size which will result in getting a good SNR. Note that increasing the sensor size beyond a point will cause the SNR to saturate, this is because some of the electric field lines from Tx/Rx electrode do not interact with the finger as shown in [Figure 313](#).

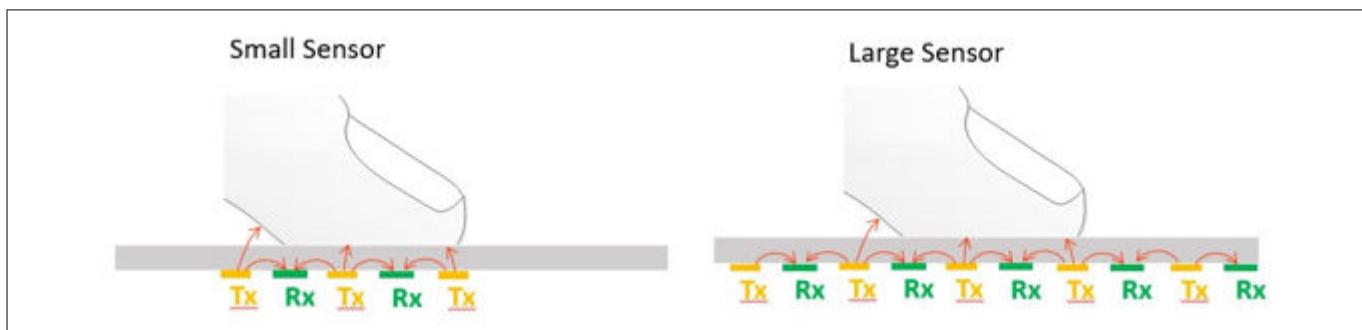


Figure 313 Interaction of electric filed with the finger

The SNR of the button is decreased with usage of thick overlays. Thus, the recommended minimum sensor size is finger size plus overlay thickness to achieve a good SNR even with thick overlays. For example, the minimum sensor size recommended could be 13x10 mm, considering the finger size around 10mm in diameter and 3 mm overlay thickness. As mentioned in the [Button design for arbitrary shapes and dimensions](#) Rx prongs should be perpendicular to the side with large dimension.

Button spacing

The button spacing is the gap between the Tx wall of two buttons. It helps to prevent user error by isolating the buttons from each other and reduces the cross talk. It is recommended to keep a minimum of 8 mm spacing between the buttons this will ensure a good single touch and multi touch performance.

Overlay

The overlay thickness and overlay permittivity influences the SNR of the button and immunity towards the external noise such as ESD. Refer to the [Overlay selection](#) section for more details. It is recommended to keep the overlay thickness as minimum as possible which will help in getting a higher SNR for the button and it

~~5 PSoC™ 6 application notes~~

should be high enough to provide immunity towards ESD noise. In some cases, if the overlay thickness is more and you cannot avoid it due to mechanical design consideration. In such a case, for getting a better SNR use a mutual capacitance button with bigger size than the recommended size. Refer to the section [Sensor size](#) for selecting the minimum button dimension with respect to the overlay thickness. Using an overlay material with higher dielectric constant will also leads to higher SNR. So always use material with high dielectric constant when we use thick overlay. And also, for smaller buttons better to have thin overlays for getting good SNR.

Air gap between Tx and Rx electrode

The gap between the Tx and Rx electrode influences the mutual capacitance between the Tx and Rx electrode. Increasing the gap reduces the mutual capacitance. It is the most critical parameter in the Fishbone pattern design and the gap between the Tx and Rx electrode such that the mutual capacitance is above 750 fF.

Number of Rx-prongs

The number of Rx prongs influence the mutual capacitance between the Tx and Rx electrode, because increasing the number of Rx prongs decreases the gap between the Tx and Rx electrode for a given button size. Higher mutual capacitance implies higher electric field lines between Tx and Rx electrode. Thus, we get a higher signal when we touch the button, because the finger touch will disturb the electric field to a maximum extent. But higher C_M also increases the impact of external noise such as VDDA ripple noise. Thus, there is a tradeoff in selecting the number of Rx prongs to get a higher signal verses getting good noise immunity. The optimal number of Rx prongs is 2 for the Fishbone pattern (that is Fishbone pattern with a single Tx prong and two Rx prongs). The below figure shows the mutual capacitance button with three and one number of Rx prongs.

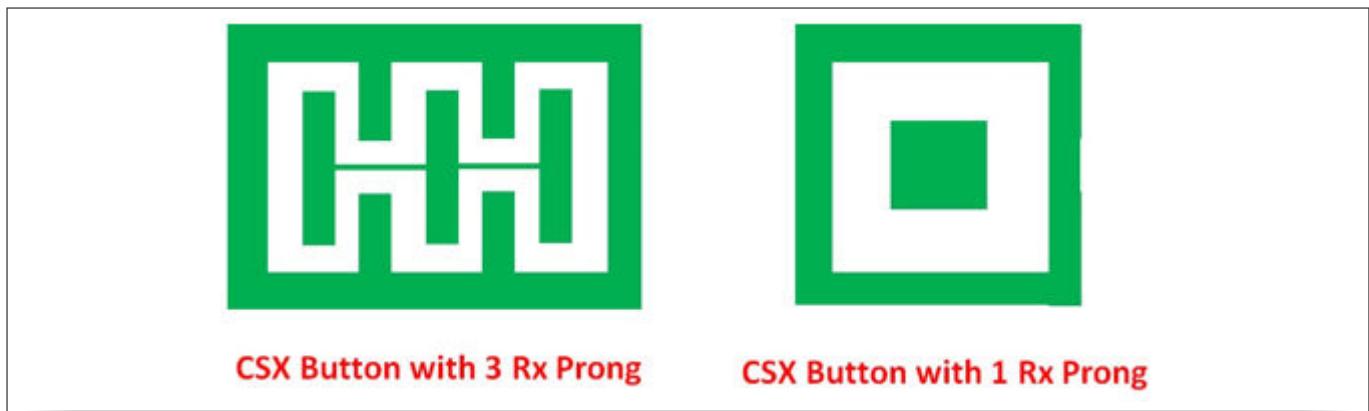


Figure 314 Mutual capacitance button with different number of prongs

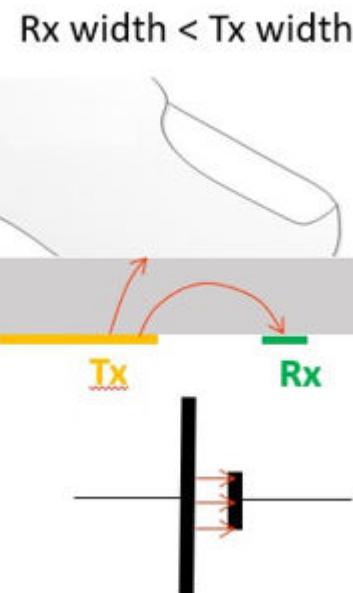
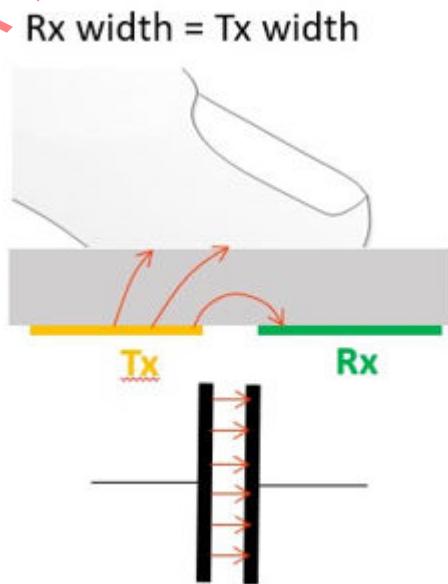
Tx electrode and Rx electrode width

The Tx electrode and Rx electrode width influences the mutual capacitance between Tx and Rx electrode. Best signal response is achieved when Rx width/area is equal to Tx width/area in the case of less external noise in the system. The below figure shows the electric filed lines from Tx to Rx electrode with equal and unequal widths. Thus, from the below figure it is clear that having equal Tx and Rx width will eventually leads to higher change in C_M for a finger touch.

Figure 315 Effect of Tx and Rx width

5 PSoC™ 6 application notes

DRAFT



In some cases, it is required to provide liquid tolerance to the mutual capacitance button as mentioned in section [Liquid tolerance for mutual-capacitance sensing](#). To achieve that we have to use a hybrid sensing technique with both CSX and CSD sensing method. In such a case the Tx or Rx electrode whichever is scanned in CSD technique should have a significant width so that it ensures a good signal for a finger touch.

Co planar ground

Presence of coplanar ground decreases the impact of noise in the system and it also provides good ground resulting in decreased signal disparity effect. It is recommended to have as much area surrounding the sensor with hatched pattern and connected it to device ground. Also follow the recommendations as mentioned in the layout and schematics guidelines in this chapter. Ground plane reduces the coupling of electric field lines to the approaching finger, which decreases the change in mutual capacitance caused by a finger touch. It is suggested to avoid having ground plane underneath the sensor unless you expect strong coupling to a noise source present right below the sensor. [Figure 309](#) shows the coplanar ground on the top and bottom layer of the PCB. The gap between the outer wall of the Tx electrode and the coplanar hatch ground should be greater than the air-gap of Tx and Rx electrodes.

Tx wall (X-wall and Y-wall width)

Tx wall act as a shield to the Rx electrode from noise. Wide Tx wall also reduces the effect of cross talk and the impact of Co-planar ground. It is recommended to keep the Tx wall width approximately equal to half of Tx electrode width. The below figure shows the effect of wider Tx wall, it increases the number of electric field lines reaching the finger from the Tx electrode by reducing the impact of Coplanar ground. The width of Tx wall can also be slightly increased in case Tx electrode is scanned as a CSD sensor as mentioned in section [Liquid tolerance for mutual-capacitance sensing](#). An example 10x10 pattern with increased Tx wall is given in [Table 63](#).

5 PSoC™ 6 application notes

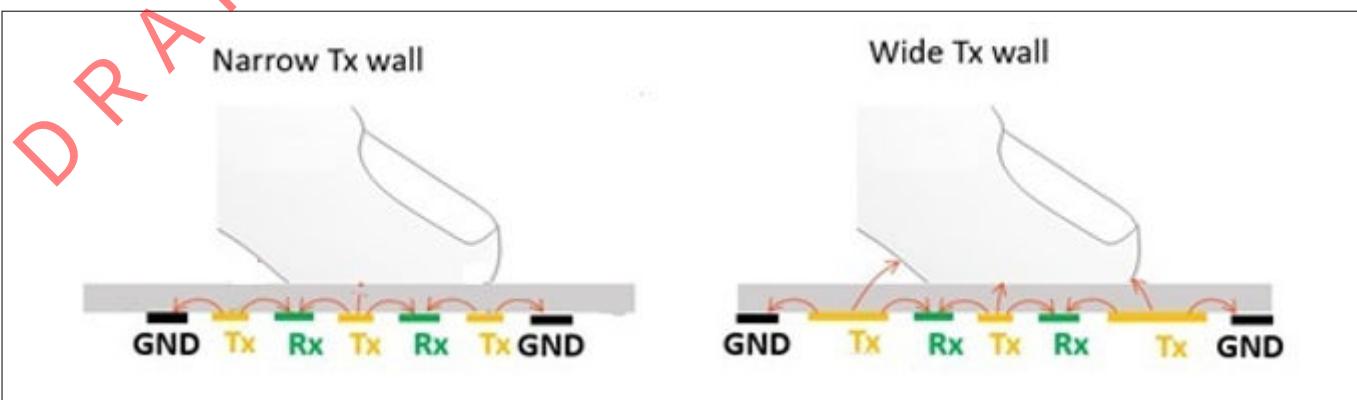


Figure 316 Effect of width of Tx wall

Table 64 Dimension of 10x10 button with increased Tx wall (all units in mm)

Button Size (X-Size, Y-Size) (mm)	Number of Rx-Prongs	Air Gap between Tx and Rx in mm	Tx Width in mm	Rx Width in mm	X-Wall Width in mm	Y-Wall Width in mm	Y Prong in mm
10, 10	2	0.8	1.2	1.2	1.5	1.6	0.2

5.8.7.4.4 Slider design

Figure 317 shows the recommended slider pattern for a linear slider and Table 65 shows the recommended values for each of the linear slider dimensions. A detailed explanation on the recommended layout guidelines is provided in the following sections.

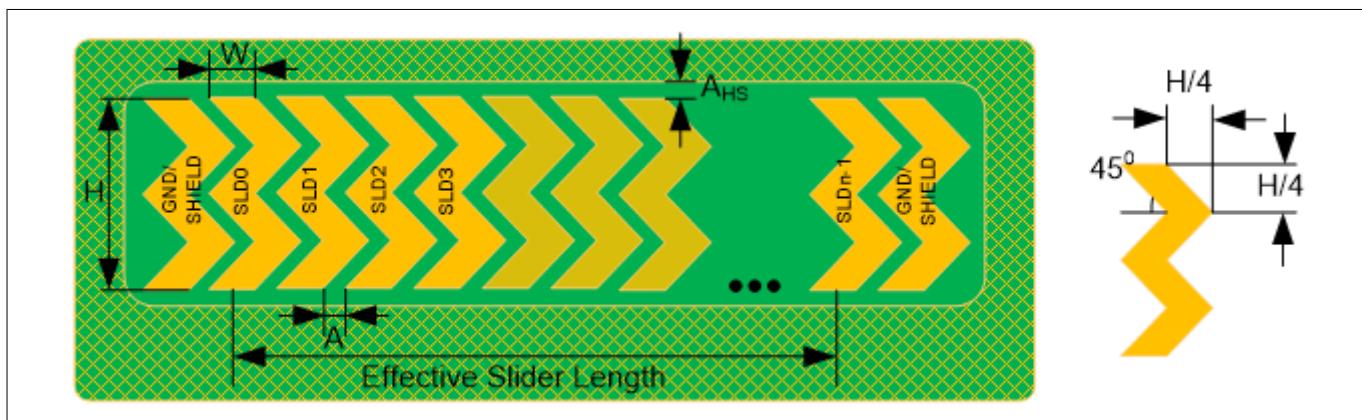


Figure 317 Typical linear slider pattern

Table 65 Linear slider dimensions

Parameter	Acrylic overlay thickness	Minimum	Maximum	Recommended
Width of the segment (W)	1 mm	2 mm	-	8 mm ²²⁾
	3 mm	4 mm	-	

(table continues...)

²² The recommended slider-segment width is based on an average human finger diameter of 9 mm. See section [Slider-segment shape, width, and Air gap](#) for more details.

DRAFT

5 PSoC™ 6 application notes

Table 65 (continued) Linear slider dimensions

Parameter	Acrylic overlay thickness	Minimum	Maximum	Recommended
	4 mm	6 mm	-	
Height of the segment (H)	-	7 mm ²³⁾	15 mm	12 mm
Air gap between segments (A)	-	0.5 mm	2 mm	0.5 mm
Air gap between the hatch and the slider (A_{HS})	-	0.5 mm	2 mm	Equal to overlay thickness

Slider-segment shape, width, and Air gap

A linear response of the reported finger position (that is, the centroid position) versus the actual finger position on a slider requires that the slider design is such that whenever a finger is placed anywhere between the middle of the segment SLD0 and middle of segment SLDn-1, other than the exact middle of slider segments, exactly two sensors report a valid signal ²⁴⁾. If a finger is placed at the exact middle of any slider segment, the adjacent sensors should report a difference count = noise threshold. Therefore, it is recommended that you use a double-chevron shape as Figure 317 shows. This shape helps in achieving a centroid response close to the ideal response, as Figure 318 and Figure 319 show. For the same reason, the slider-segment width and air gap (dimensions "W" and "A" respectively, as marked in Figure 317) should follow the relationship mentioned in Equation 39.

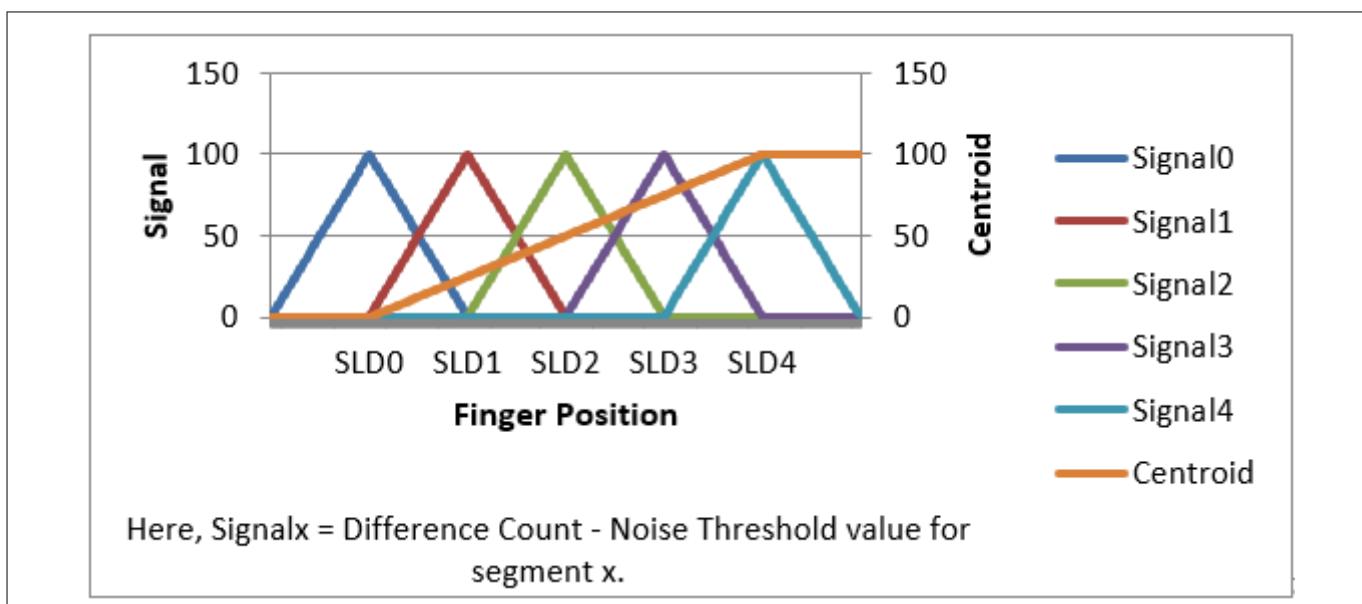


Figure 318 Ideal slider segment signals and centroid response

²³ The minimum slider segment height of 7 mm is recommended based on a minimum human finger diameter of 7 mm. Slider height may be kept lower than 7 mm if the overlay thickness and CAPSENSE™ tuning is such that an [Signal-to-noise ratio \(SNR\)](#) (SNR) $\geq 5:1$ is achieved when the finger is placed in the middle of any segment.

²⁴ Here, a valid signal means that the difference count of the given slider segment is greater than or equal to the noise threshold value.

5 PSoC™ 6 application notes

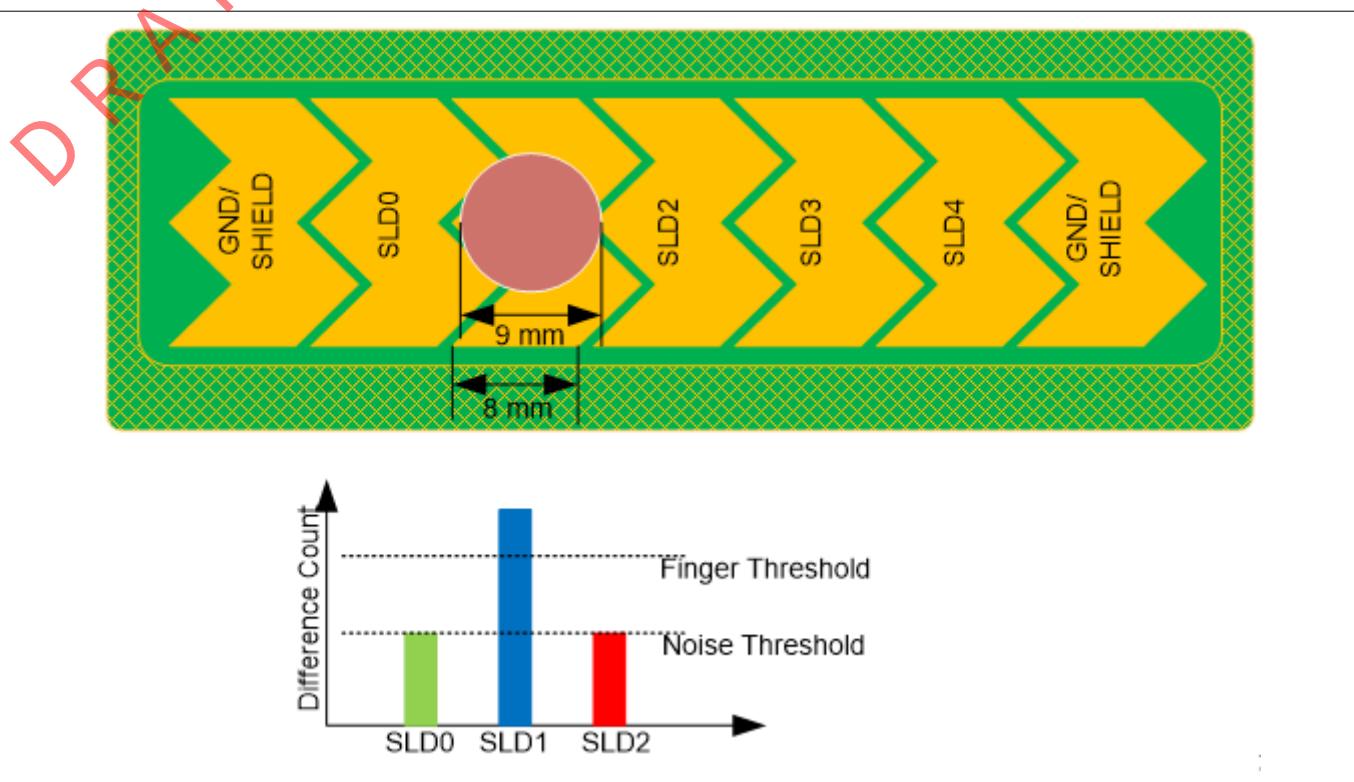


Figure 319 Ideal slider signals

$$W+2A = \text{finger diameter}$$

Equation 74 Segment width and air gap relation with the finger diameter

Typically, an average human finger diameter is approximately 9 mm. Based on this average finger diameter and [Equation 74](#), the recommended slider-segment-width and air-gap is 8 mm and 0.5 mm respectively.

If the sum of *slider-segment width* and $2 * \text{air-gap}$ is lesser than *finger diameter*, as required according to [Equation 74](#), the centroid response will be non-linear. This is because, in this case, a finger placed on the slider will add capacitance, and hence valid signal to more than two slider-segments at some given position, as [Figure 320](#) shows. Thus, calculated centroid position in [Equation 75](#) will be non-linear as [Figure 321](#) shows.

5 PSoC™ 6 application notes

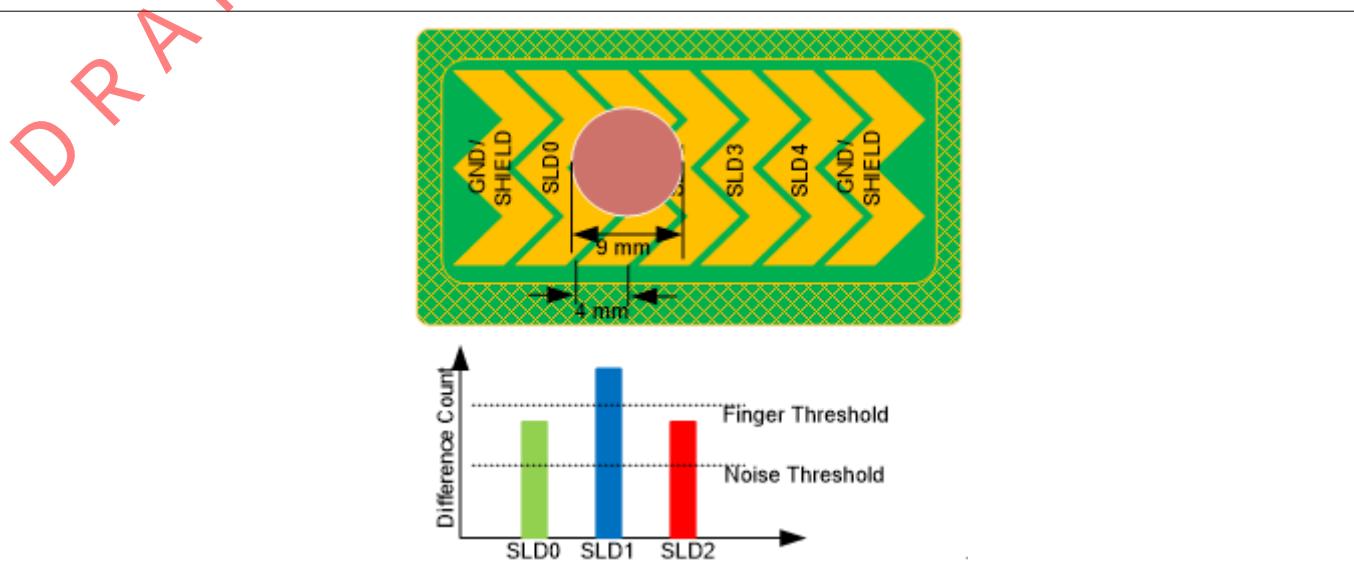


Figure 320 Finger causes valid signal on more than two segments when slider segment width is lower than recommended

$$\text{centroid position} = \left(\frac{S_{x+1} - S_{x-1}}{S_{x+1} + S_x + S_{x-1}} + x \right) \times \frac{\text{Resolution}}{(n-1)}$$

Equation 75 Centroid algorithm used by CAPSENSE™ component in PSoC™ Creator

Where,

Resolution = API resolution set in the CAPSENSE™ component customizer

n = Number of sensor elements in the CAPSENSE™ component customizer

x = Index of element which gives maximum signal

Si = Different counts (with subtracted noise threshold value) of the slider segment

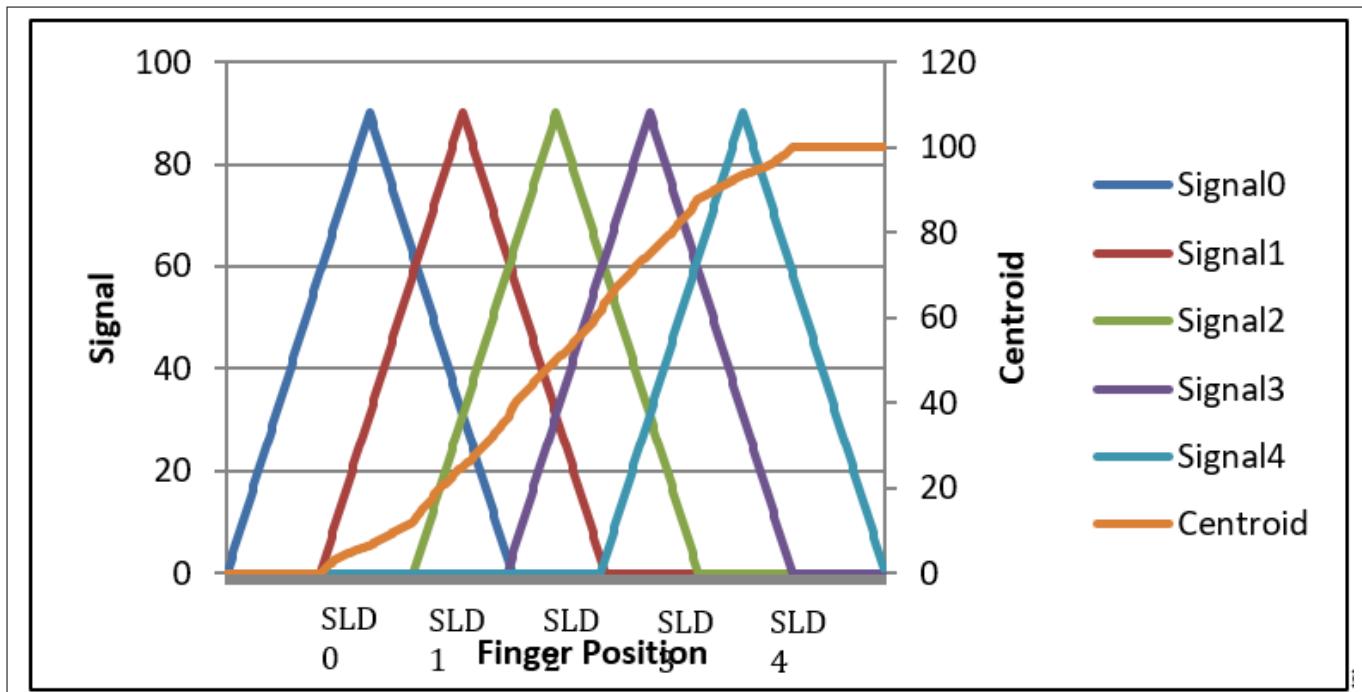


Figure 321 Nonlinear centroid response when slider segment width is lower than recommended

5 PSoC™ 6 application notes

~~DRAFT~~

Note that even though a *slider-segment-width* value of less than *finger diameter* – 2 * *air-gap* provides a non-linear centroid response, as Figure 321 shows; it may still be used in an end application where the linearity of reported centroid versus actual finger position does not play a significant role. However, a minimum value of slider-segment-width must be maintained, based on overlay thickness, such that, at any position on the effective slider length, at least one slider-segment provides a **Signal-to-noise ratio (SNR)** of $\geq 5:1$ (that is signal greater than or equal to the finger threshold parameter) at that position. If the slider-segment width is too low, a finger may not be able to couple enough capacitance, and therefore, none of the slider-segments will have a 5:1 SNR, resulting in a reported centroid value of 0xFFFF²⁵ in PSoC™ Creator as Figure 322 shows, and 0x0000²⁶ in ModusToolbox™.

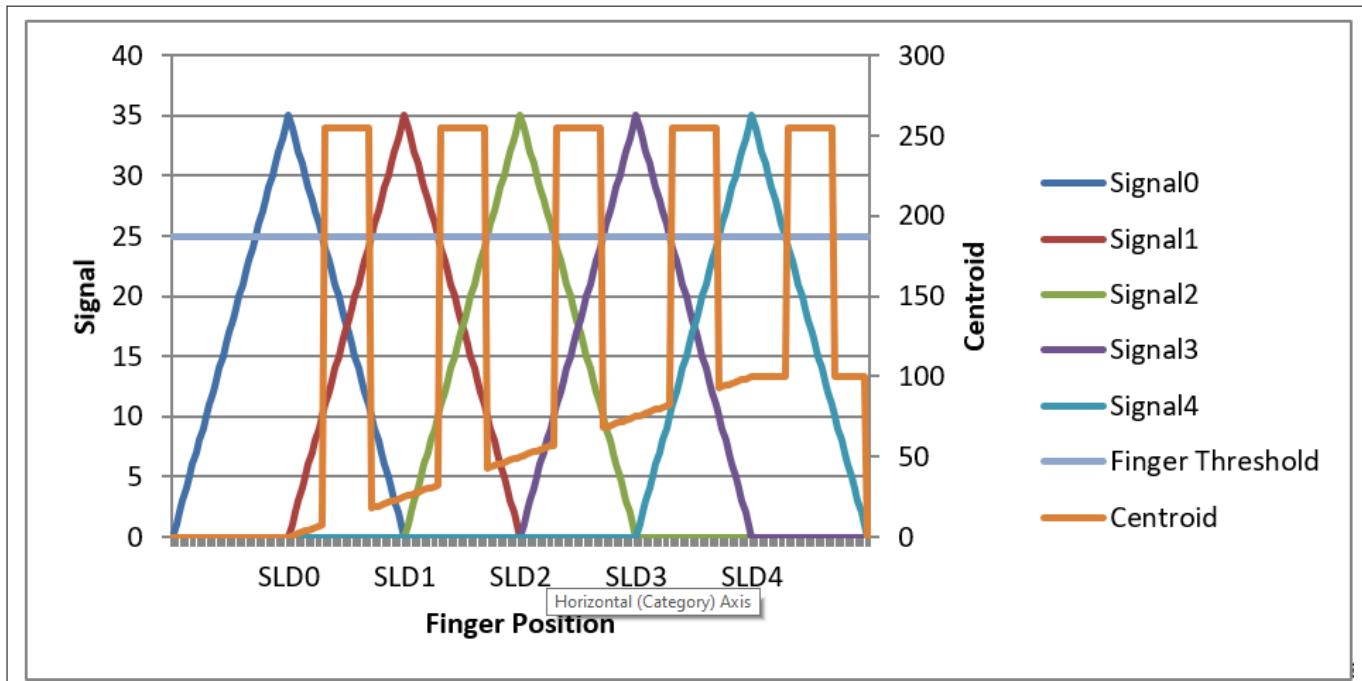


Figure 322 Incorrect centroid reported when slider-segment-width is too low

The minimum value of slider-segment width for certain overlay thickness values for an acrylic overlay are provided in Table 65. For thickness values of acrylic overlays, which are not specified in Table 65, Figure 323 may be used to estimate the minimum slider-segment width. Very thin overlay or no overlay may cause a nonlinear centroid response due to saturation of raw count or due to high finger capacitance; centroid position may be detected before touching the slider. In these conditions, the CAPSENSE™ centroid algorithm will not be able to correctly estimate the finger position on the slider using Equation 75. It is recommended to have the overlay thickness for the CSD sensor as mentioned in #unique_733/unique_733_Connect_42_table_g1d_vz2_ftb.

²⁵ The CAPSENSE™ Component in PSoC™ Creator reports a centroid of 0xFFFF when there is no finger detected on the slider, or when none of the slider segments reports a difference count value greater than the Finger Threshold parameter.

²⁶ The CAPSENSE™ middleware in ModusToolbox™ reports a centroid of 0x0000 when there is no finger detected on the slider, or when none of the slider segments reports a difference count value greater than the Finger Threshold parameter.

5 PSoC™ 6 application notes

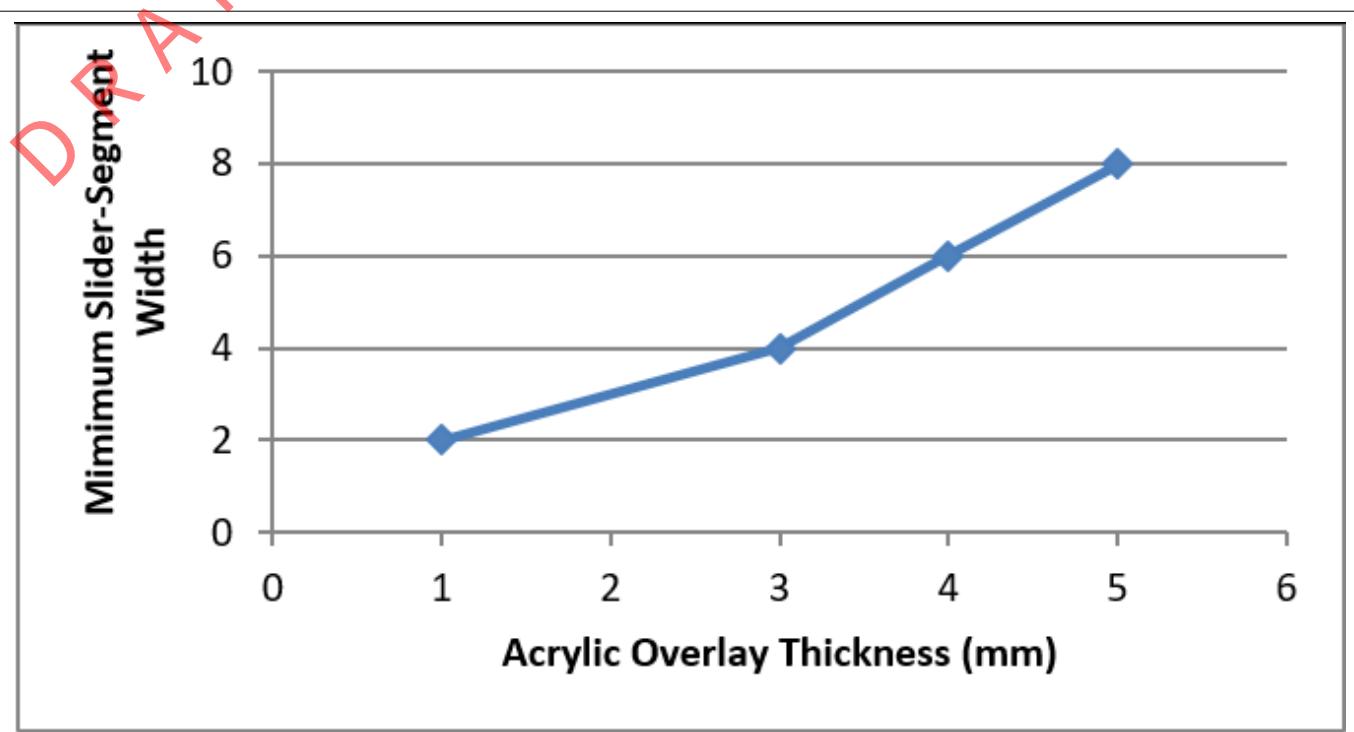


Figure 323 Minimum slider-segment width with respect to overlay thickness for an acrylic overlay

If the *slider-segment-width + 2 * air-gap* is higher than the *finger diameter* value as required in [Equation 74](#), the centroid response will have flat-spots; that is, if the finger is moved towards the middle of any segment, the reported centroid position will remain constant as [Figure 324](#) shows. This is because, as [Figure 325](#) shows, when the finger is placed in the middle of a slider segment, it will add a valid signal only to that segment even if the finger is moved a little towards adjacent segments.

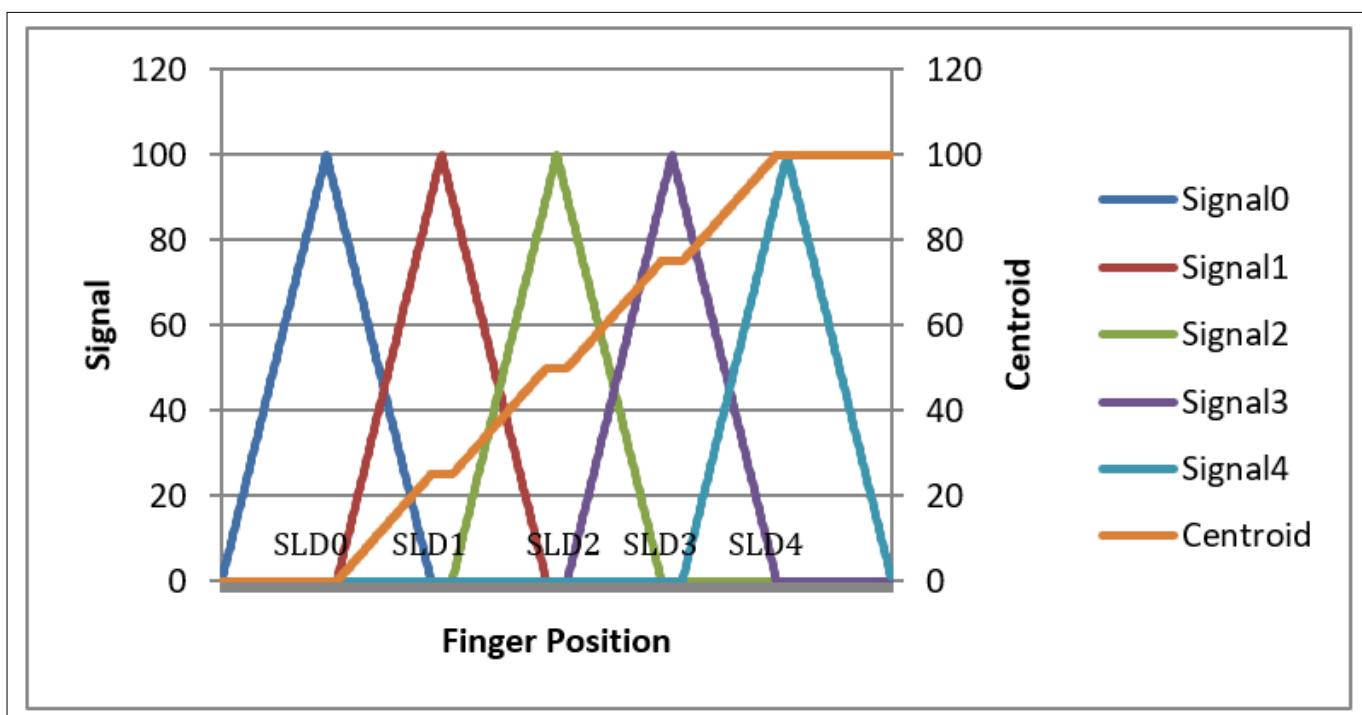


Figure 324 Flat-spots (nonresponsive centroid) when slider-segment width is higher than recommended

5 PSoC™ 6 application notes

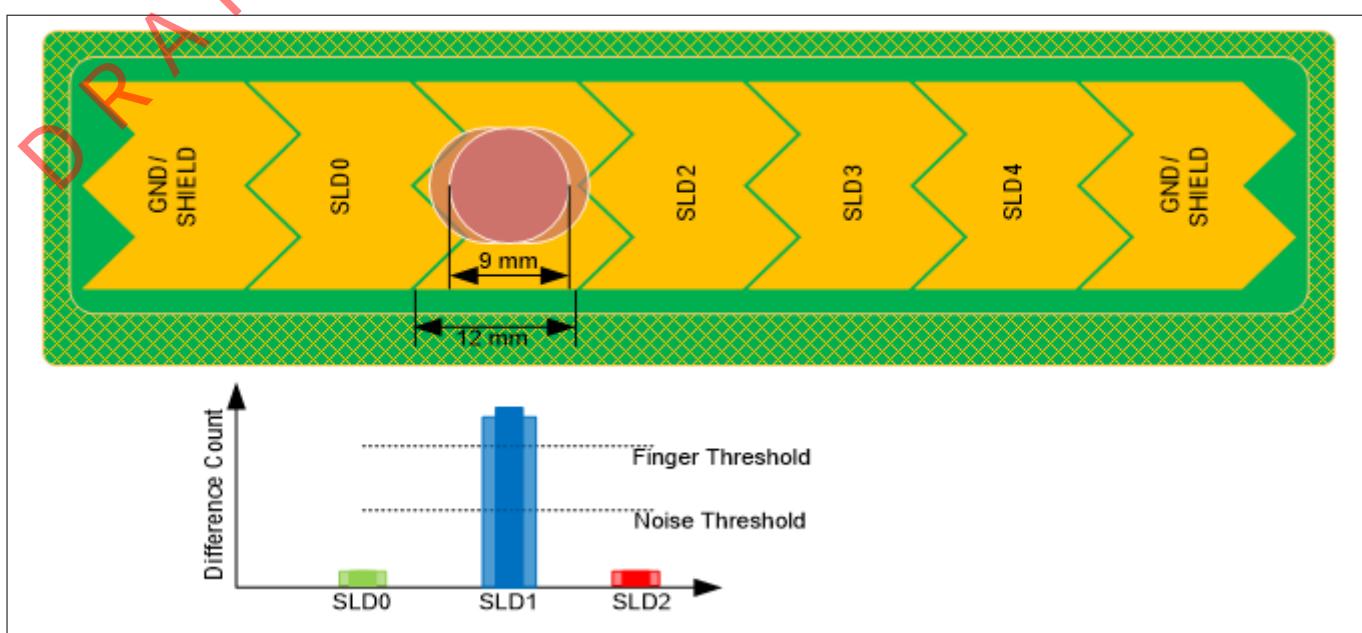


Figure 325 Signal on slider segments when slider-segment width is higher than recommended

Note that if the value of *slider-segment-width* + 2 * *air-gap* is higher than the *finger diameter*, it may be possible to increase and adjust the sensitivity of all slider segments such that even if the finger is placed in the middle of a slider segment, adjacent sensors report a difference count value equal to the noise threshold value (see [Figure 318](#)); however, this will result in the hover effect – the slider may report a centroid position even if the finger is hovering above the slider and not touching the slider.

Dummy segments at the ends of a slider

In a CAPSENSE™ design, when one segment is scanned, adjacent segments are connected to either ground or to the driven-shield signal based on the option specified in the “Inactive sensor connection” parameter in the CAPSENSE™ CSD Component. For a linear centroid response, the slider requires all the segments to have the same sensitivity, that is, the increase in the raw count (signal) when a finger is placed on the slider segment should be the same for all segments. To maintain a uniform signal level from all slider segments, it is recommended that you physically connect the two segments at both ends of a slider to either ground or driven shield signal. The connection to ground or to the driven-shield signal depends on the value specified in the “Inactive sensor connection” parameter. Therefore, if your application requires an ‘n’ segment slider, it is recommended that you create n+2 physical segments, as [Figure 317](#) shows.

If it is not possible to have two segments at both ends of a slider due to space constraints, you can implement these segments in the top hatch fill, as [Figure 326](#) shows. Also, if the total available space is still constrained, the width of these segments may be kept lesser than the width of segments SLD0 through SLDn-1, or these dummy segments may even be removed.

If the two segments at the both ends of a slider are connected to the top hatch fill, you should connect the top hatch fill to the signal specified in the “Inactive sensor connection” parameter. If liquid tolerance is required for the slider, the hatch fill around the slider, the last two segments, and the inactive slider segments should be connected to the driven-shield signal. See the [Effect of liquid droplets and liquid stream on a self-capacitance sensor](#) section for more details.

5 PSoC™ 6 application notes



Figure 326 Linear slider pattern when first and last segments are connected to top hatch fill

Deciding slider dimensions

Slider dimensions for a given design can be chosen based on following considerations:

1. Decide the required length of the slider (L) based on application requirements. This is same as the “effective slider length” as [Figure 317](#) shows.
2. Decide the height of the segment based on the available space on the board. Use the maximum allowed segment height (15 mm) if the board space permits; if not, use a lesser height but ensure that the height is greater than the minimum specified in [Table 65](#).
3. The slider-segment width and the air gap between slider segments should be as recommended in [Table 65](#). The recommended slider-segment-width and air-gap for an average finger diameter of 9 mm is 8 mm and 0.5 mm respectively.
4. For a given slider length L, calculate the number of segments required by using the following formula:

$$\text{Number of segments} = \frac{\text{slider length}}{\text{slider segment width} + \text{airgap}} + 1$$

Equation 76 Number of segments required for a slider

Note: *A minimum of two slider segments are required to implement a slider.*

If the available number of CAPSENSE™ pins is slightly less than the number of segments calculated for a certain application, you may increase the segment width to achieve the required slider length with the available number of pins. For example, a 10.2 cm slider requires 13 segments. However, if only 10 pins are available, the segment width may be increased to 10.6 cm. This will either result in a nonlinear response as [Figure 324](#) shows, or a hover effect; however, this layout may be used if the end application does not need a high linearity.

Note: *The PCB length is higher than the required slider length as [Figure 317](#) shows. PCB length can be related to the slider length as shown in [Equation 77](#).*

$$\text{PCB length} = \text{Slider length} + 3 \times \text{slider segment width} + 2 \times \text{air gap}$$

Equation 77 Relationship between minimum PCB length and slider length

If the available PCB area is less than that required per this equation, you can remove the dummy segments. In this case, the minimum PCB length required will be as shown in [Equation 78](#).

5 PSoC™ 6 application notes

~~DRAFT~~
PCB length = *Slider length* + *Slider segment width*

Equation 78 Relationship between minimum PCB length and slider length

Routing slider segment trace

A slider has many segments, each of which is connected separately to the CAPSENSE™ input pin of the device. Each segment is separately scanned and the centroid algorithm is applied finally on the signal values of all the segments to calculate the centroid position. The SmartSense algorithm implements a specific tuning method for sliders to avoid nonlinearity in the centroid that could occur due to the difference of C_p in the segments. However, the following layout conditions need to be met for the slider to work:

1. C_p of any segment should always be within the supported range of C_p as mentioned in the [Component datasheet](#)
2. C_p of the slider segment should be as close as possible. However, in the practical scenario C_p of each slider segment might vary because of differences in trace routing for each segment. The maximum allowed variation in the segment parasitic capacitance is 44% maximum C_p of the slider segment for an 85% IDAC calibration level. If the variation in C_p is beyond this limit then it may cause a change in the sensitivity among the slider segments leading to a non-linear slider response

Implement the following layout design rules to meet a good slider design with linear response.

- Design the shape of all segments to be as uniform as possible
- Ensure that the length and the width of the traces connecting the segments to the device are same for all the segments if possible
- Maintain the same air gap between the sensors or traces to ground plane or hatch fill

Slider design with LEDs

In some applications, it may be required to display the finger position by driving LEDs. You can either place the LEDs just above the slider segments or drill a hole in the middle of a slider segment for LED backlighting, as [Figure 327](#) shows. When a hole is drilled for placing an LED, the effective area of the slider segment reduces. To achieve an $SNR > 5:1$, you need to have a slider segment with a width larger than the LED hole size. See [Table 65](#) for the minimum slider width required to achieve an $SNR > 5:1$ for a given overlay thickness. Follow the guidelines provided in the [Crosstalk solutions](#) section to route the LED traces.

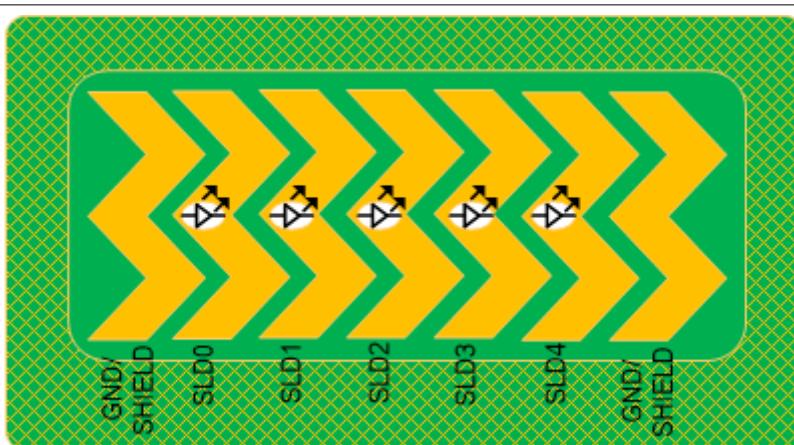


Figure 327 Slider design with LED backlighting

5 PSoC™ 6 application notes~~DRAFT~~
5.8.7.4.5 Sensor and device placement

Follow these guidelines while placing the sensor and the device in your PCB design:

- Minimize the trace length from the device pins to the sensor pad
- Mount series resistors within 10 mm of the device pins to reduce RF interference and provide ESD protection. See [Series resistors on CAPSENSE pins](#) for details
- Mount the device and the other components on the bottom layer of the PCB
- Avoid connectors between the sensor and the device pins because connectors increase C_p and noise pickup
- Button to Button distance (edge to edge) must be greater than 8 mm. If keys have less than 8 mm between them, there will be cross talk between the keys. Also, from a usability standpoint, it increases the risk of the user touching two keys at the same time. Key to key distance must be greater than 8 mm
- Spacing from a touch line to any metal should be greater than 5 mm. This includes the metal chassis, decorative chrome trim, screws, and so on
- Isolate or provide physical separation between CAPSENSE™ components and their signals from noisy subsystems such as transformers. A CAPSENSE™ system in general is sensitive to external noise

5.8.7.4.6 Trace length and width

Use short and narrow PCB traces to minimize the parasitic capacitance of the sensor. The maximum recommended trace length is 12 inches (300 mm) for a standard PCB and 2 inches (50 mm) for flex circuits. The maximum recommended trace width is 7 mil (0.18 mm). You should surround the CAPSENSE™ traces with a hatched ground or hatched shield with trace-to-hatch clearance of 10 mil to 20 mil (0.25 mm to 0.51 mm).

5.8.7.4.7 Trace routing

You should route the sensor traces on the bottom layer of the PCB, so that the finger does not interact with the traces. Do not route traces directly under any sensor pad unless the trace is connected to that sensor.

Do not run capacitive sensing traces closer than 0.25 mm to switching signals or communication lines. Increasing the distance between the sensing traces and other signals increases the noise immunity. If it is necessary to cross communication lines with sensor pins, make sure that the intersection is at right angles, as [Figure 328](#) shows.

5 PSoC™ 6 application notes

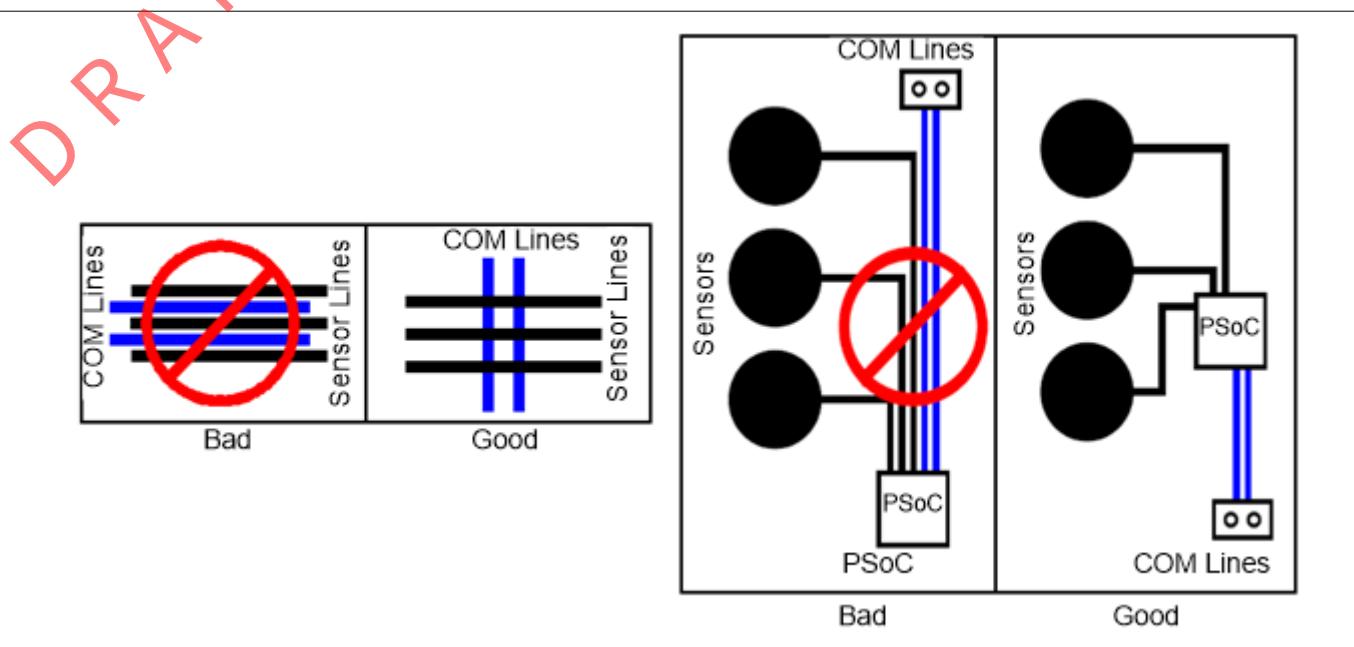


Figure 328 Routing of sensor and communication lines

If, due to spacing constraints, sensor traces run in parallel with high-speed traces such as I²C communication lines or Bluetooth® LE antenna traces, it is recommended to place a ground trace between the sensor trace and the high-speed trace as shown in [Figure 329](#). This guideline also applies to the cross talk caused by CAPSENSE™ sensor trace with precision analog trace such as traces from temperature sensor to the PSoC™ device. The thickness of the ground trace can be 7 mils and the spacing from sensor trace to ground trace should be equal to minimum of 10 mils to reduce the C_p of the CAPSENSE™ sensor.

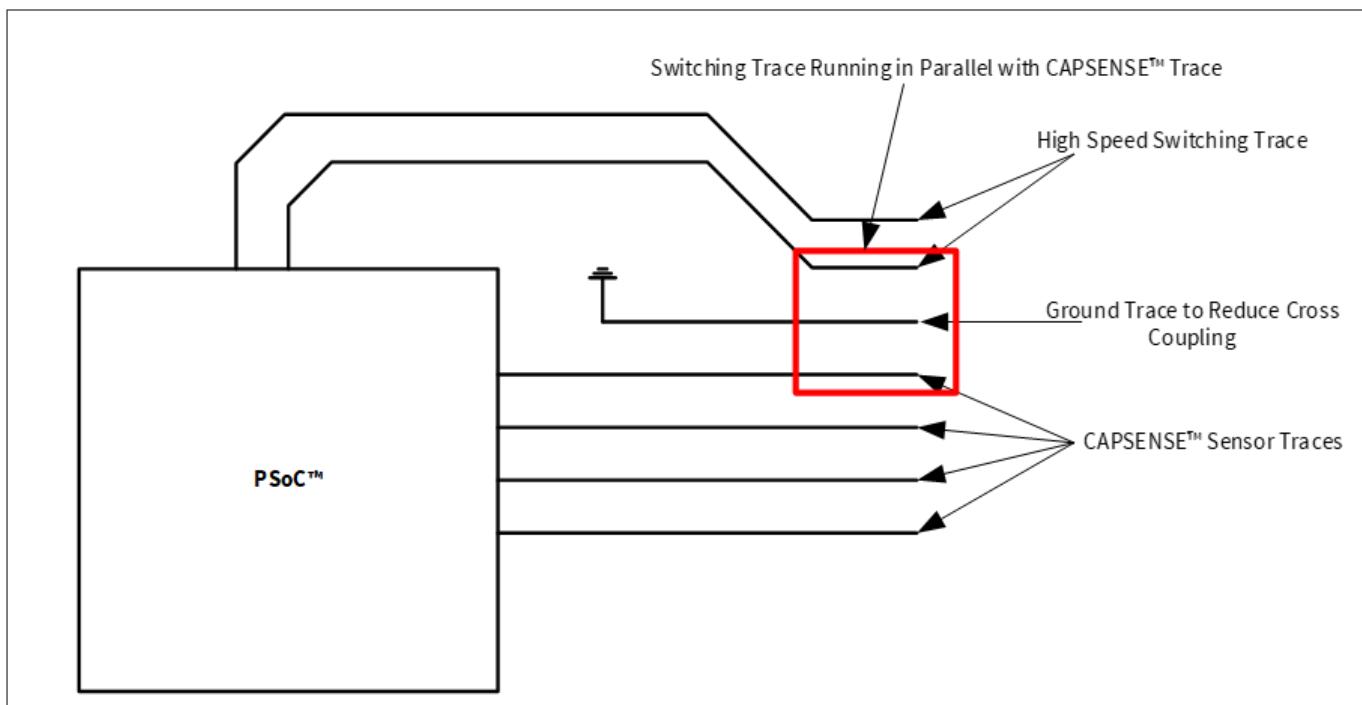


Figure 329 Reducing cross talk between high-speed switching trace and CAPSENSE™ trace

If a ground trace cannot be placed in between the switching trace and the CAPSENSE™ trace, the 3W rule can be followed to reduce the cross talk between the traces. The 3W rule states that “to reduce cross talk from

5 PSoC™ 6 application notes

adjacent traces, a minimum spacing of two trace widths should be maintained from edge to edge" as shown in Figure 330.

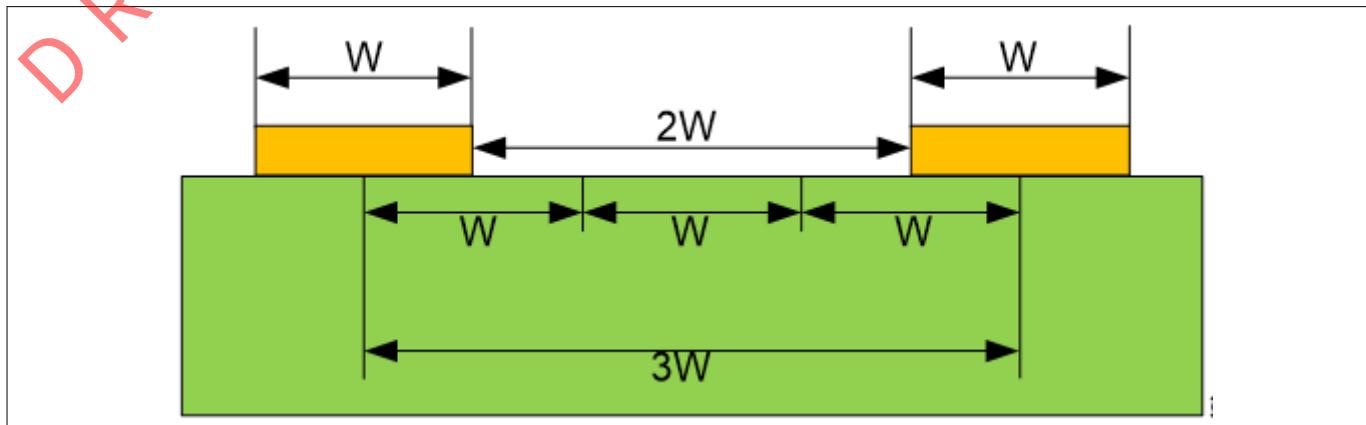


Figure 330 3W trace spacing to minimize cross talk

- Do not run Tx and Rx lines parallel to each other. The trace routing should be separated as much as possible
- If the layout constraints require Tx and Rx run parallel for short distances, the space between Tx and Rx should be greater than the distance between Tx and Rx inside the key (2 times the Tx-Rx key spacing is preferred) or add ground between them
- Keep as much clearance around Rx as possible to prevent noise on the touch keys. It is critical to follow this guideline for spacing to power traces and LED lines (high speed switching, power). Ground should also follow this rule, but it is less critical. Ground will provide noise protection but will reduce key sensitivity
- For a given set of sensors, the number of Rx lines must be less than or equal to Tx lines. Rx lines are susceptible to noise, whereas Tx lines are relatively less susceptible

5.8.7.4.8 Crosstalk solutions

A common backlighting technique for panels is an LED mounted under the sensor pad so that it is visible through a hole in the middle of the sensor pad. When the LED is switched ON or OFF, voltage transitions on the LED trace can create crosstalk in the capacitive sensor input, creating noisy sensor data. To prevent this crosstalk, isolate CAPSENSE™ and the LED traces from one another as [Trace routing](#) section explains.

You can also reduce crosstalk by removing the rapid transitions in the LED drive voltage, by using a filter as [Figure 331](#) shows. Design the filter based on the required LED response speed.

5 PSoC™ 6 application notes

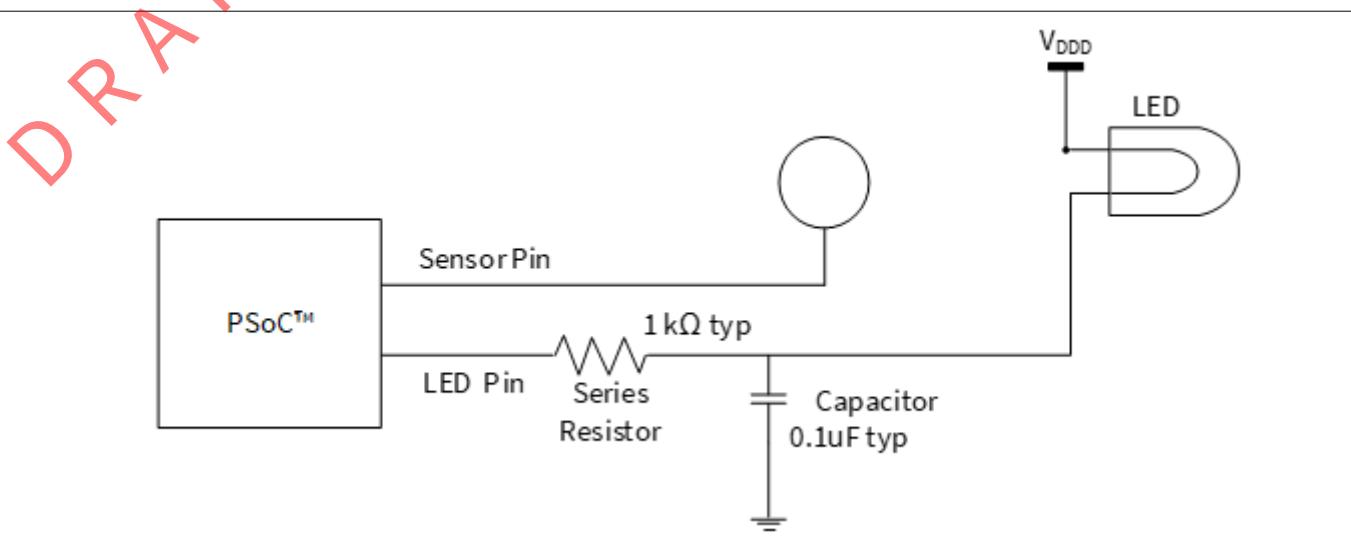


Figure 331 Reducing crosstalk

A guard trace is a ground trace running close to or above/below a TX/Rx line of a mutual-capacitance button. Guard traces can be used to protect sensor traces from noise if the layout does not allow for a ground hatch. Similar to ground hatch, guard traces add parasitic capacitance and reduce button sensitivity. Guard traces are usually needed on a case-by-case basis. Typical situations where guard traces have been used in the past include:

- Reduce cross talk
- Protect from noise of high-speed lines (I2C, SPI, UART) and toggling LED traces
- Border around the HMI or around an LCD

5.8.7.4.9 Vias

Use the minimum number of vias possible to route CAPSENSE™ signals, to minimize parasitic capacitance. Place the vias on the edge of the sensor pad to reduce trace length, as [Figure 332](#) shows.

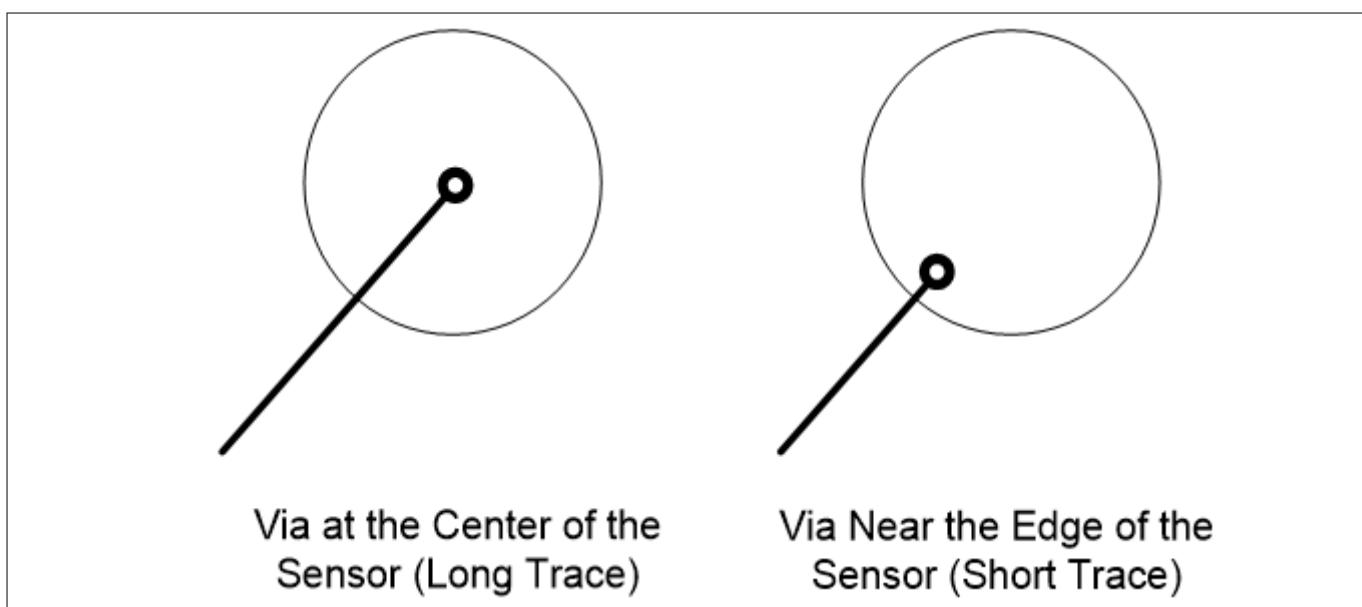


Figure 332 Via placement on the sensor pad

~~DRAFT~~ 5 PSoC™ 6 application notes

5.8.7.4.10 Ground plane

When designing the ground plane, follow these guidelines:

- Ground surrounding the sensors should be in a hatch pattern. If you are using ground or driven-shield planes in both top and bottom layers of the PCB, you should use a 25 percent hatching on the top layer (7-mil line, 45-mil spacing), and 17 percent on the bottom layer (7 mil line, 70 mil spacing)
- For the other parts of the board not related to CAPSENSE™, solid ground should be present as much as possible
- The ground planes on different layers should be stitched together as much as possible, depending on the PCB manufacturing costs. Higher amount of stitching results in lower ground inductance, and brings the chip ground closer to the supply ground. This is important especially when there is high current sinking through the ground, such as when the radio is operational
- Every ground plane used for CAPSENSE™ should be star-connected to a central point, and this central point should be the sole return path to the supply ground. Specifically:
 - The hatch ground for all sensors must terminate at the central point
 - The ground plane for C_{MOD} , C_{INTX} must terminate at the central point
 - The ground plane for C_{SH_TANK} must terminate at the central point

Figure 333 explains the star connection. The central point for different families is mentioned in [Table 66](#).

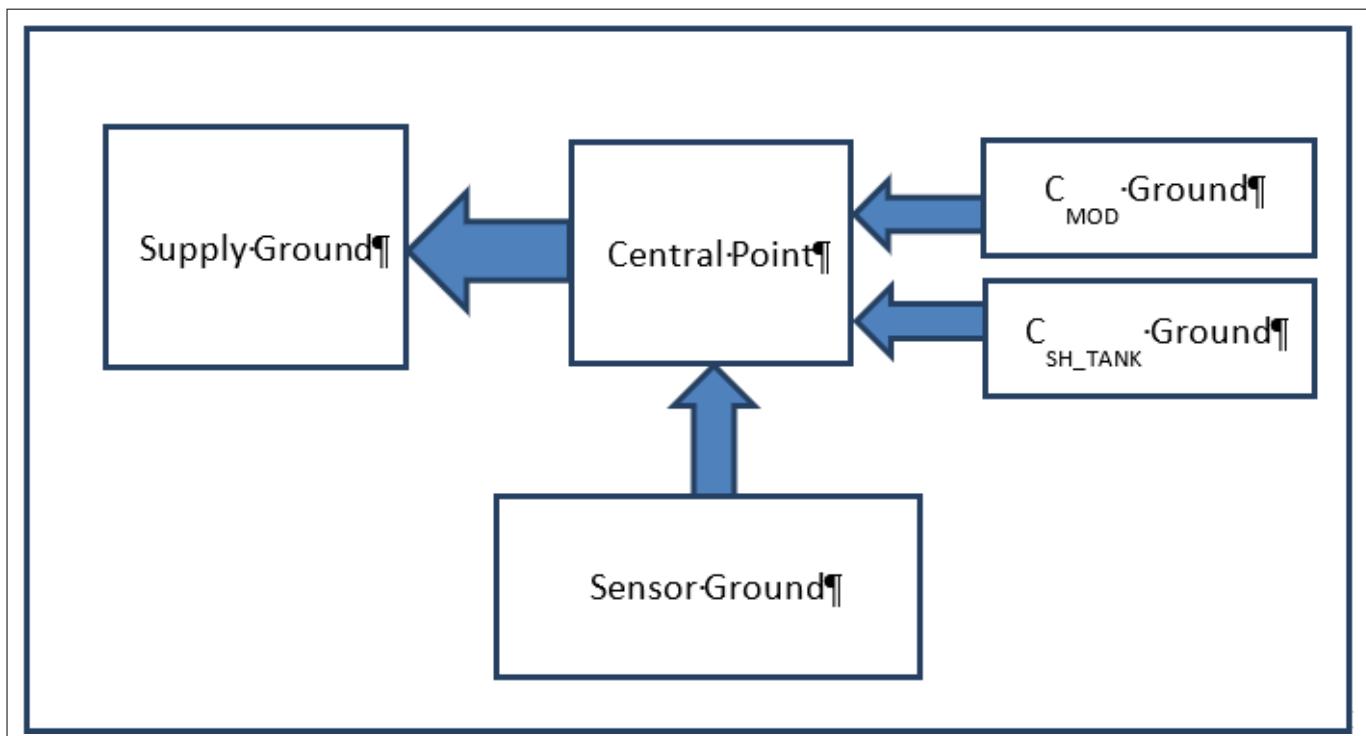


Figure 333 Star connection for Ground

Table 66 Central point for star connection

Family	Central point
PSoC™ 4000	VSS pin
PSoC™ 4100/4100M	VSS pin
PSoC™ 4200/4200M/4200L/PSoC™ 4-S/PSoC™ 4100PS	VSS pin
PSoC™ 4100-BL	E-pad

(table continues...)

DRAFT

5 PSoC™ 6 application notes

Table 66 (continued) Central point for star connection

Family	Central point
PSoC™ 4200-BL	E-pad

- All the ground planes for CAPSENSE™ should have an inductance of less than 0.2 nH from the central point. To achieve this, place the C_{MOD} , C_{INTx} , and C_{SH_TANK} capacitor pads close to the chip, and keep their ground planes thick enough

Using packages without E-pad

When not using the E-pad, the VSS pin should be the central point and the sole return path to the supply ground. High-level layout diagrams of the top and bottom layers of a board when using a chip without the E-pad are shown in [Figure 334](#) and [Figure 335](#).

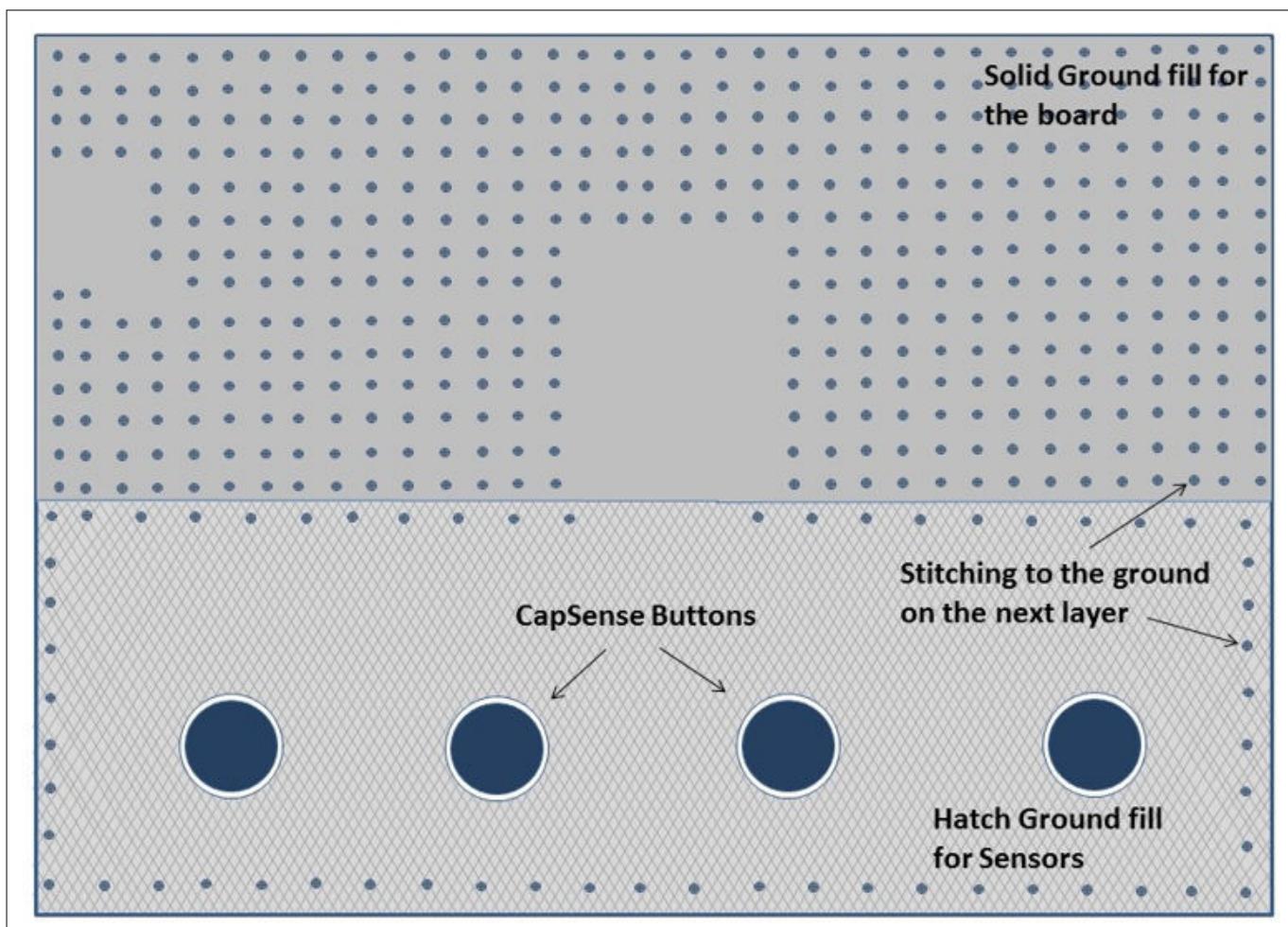


Figure 334 PCB top layer layout using a chip without E-pad

5 PSoC™ 6 application notes

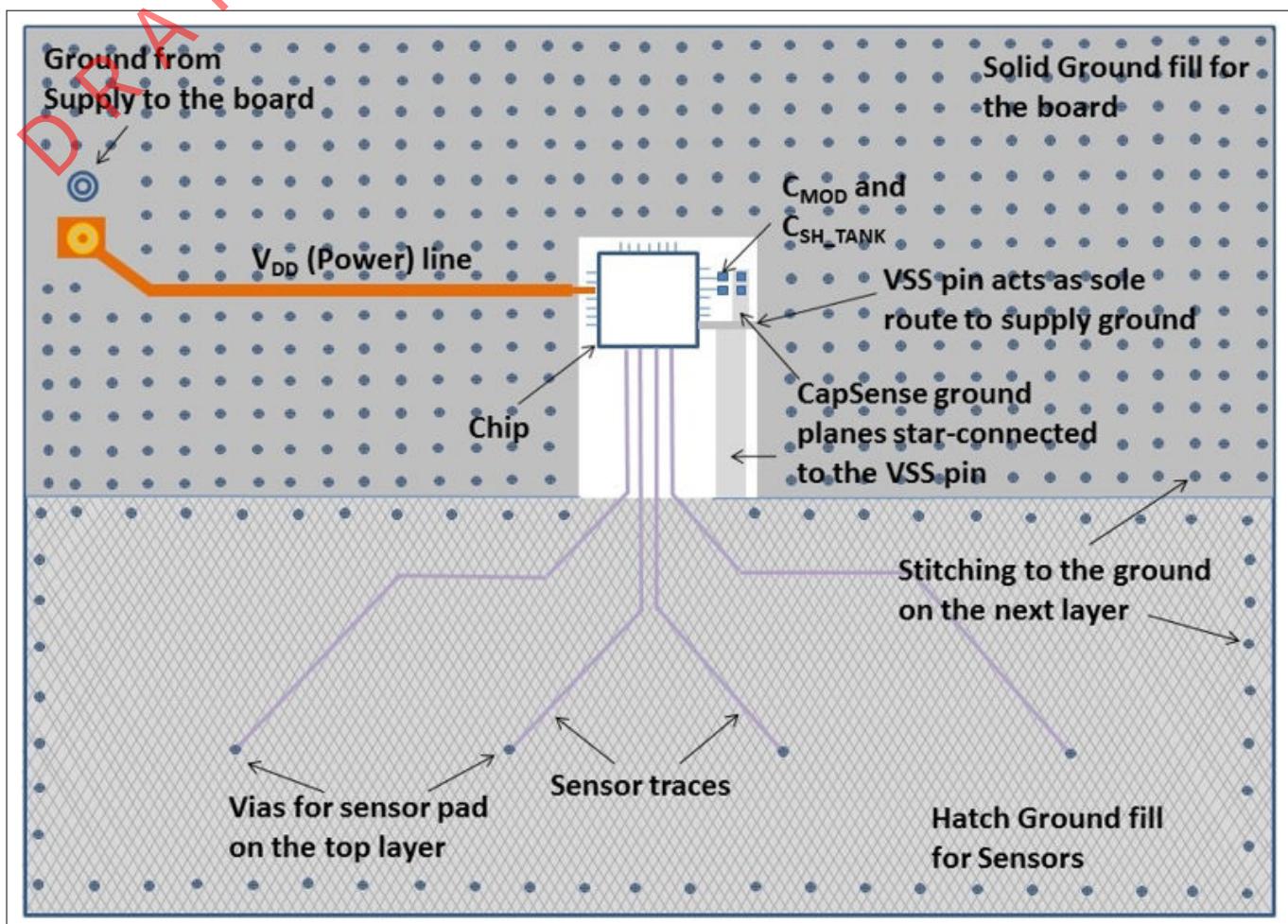


Figure 335 PCB bottom layer layout using a chip without E-pad

Using packages with E-pad

If you are using packages with E-pad, the following guidelines must be followed:

- The E-pad must be the central point and the sole return path to the supply ground
- The E-pad must have vias underneath to connect to the next layers for additional grounding. Generally unfilled vias are used in a design for cost purposes, but silver-epoxy filled vias are recommended for the best performance as they result in the lowest inductance in the ground path

Using PSoC™ 4 Bluetooth® LE devices

In the case of PSoC™ 4 Bluetooth® LE devices in the QFN package (with E-pad):

- The general guidelines of ground plane (discussed above) apply
- The E-pad usage guidelines of [Using packages with E-pad](#) apply
- The VSSA pin should be connected to the E-pad below the chip itself
- The vias underneath the E-pad are recommended to be 5x5 vias of 10-mil size

High-level layout diagrams of the top and bottom layers of a board when using PSoC™ 4 Bluetooth® LE chips are shown in [Figure 336](#) and [Figure 337](#).

5 PSoC™ 6 application notes

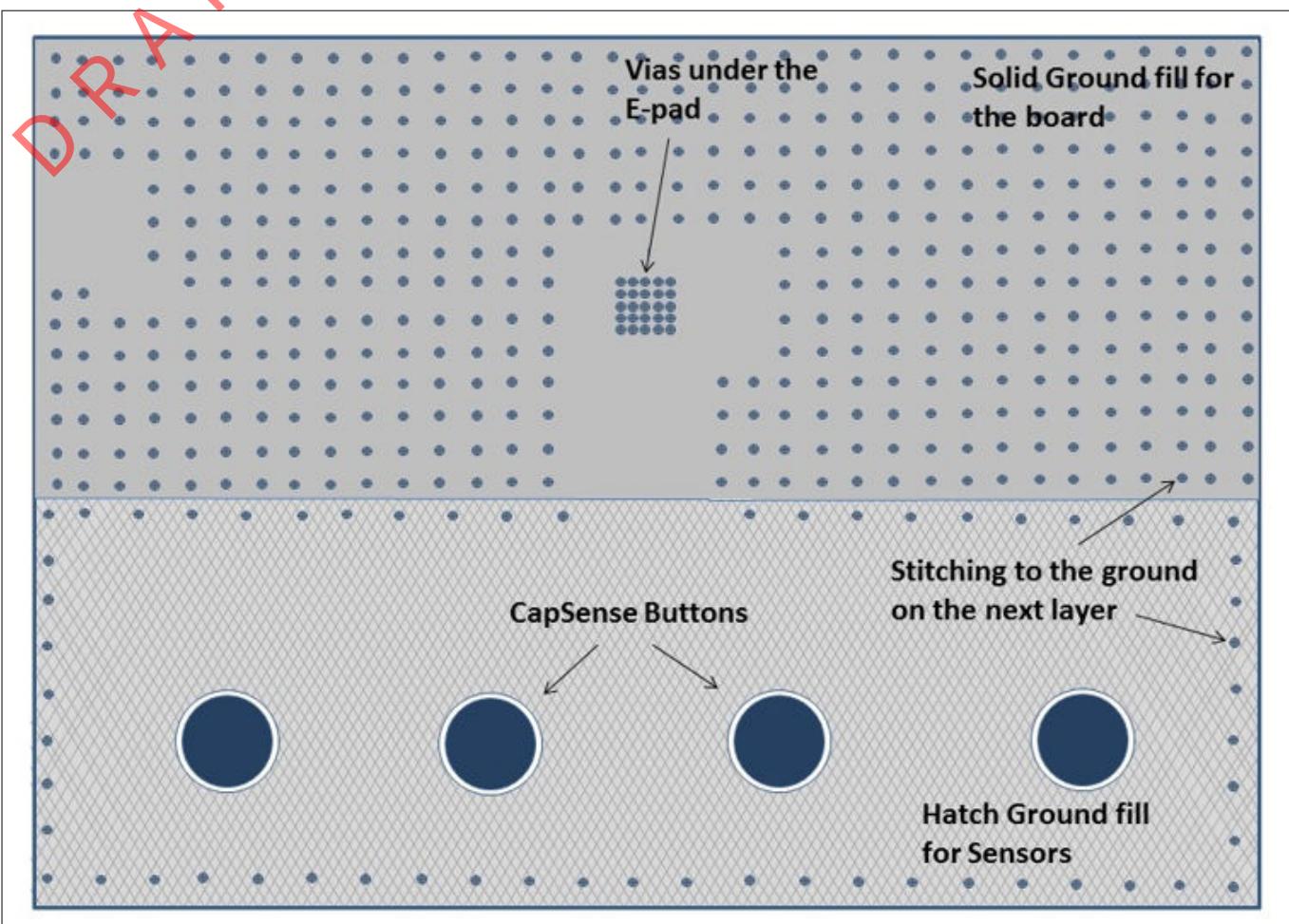


Figure 336 PCB top layer layout with PSoC™ 4Bluetooth® LE (with E-pad)

5 PSoC™ 6 application notes

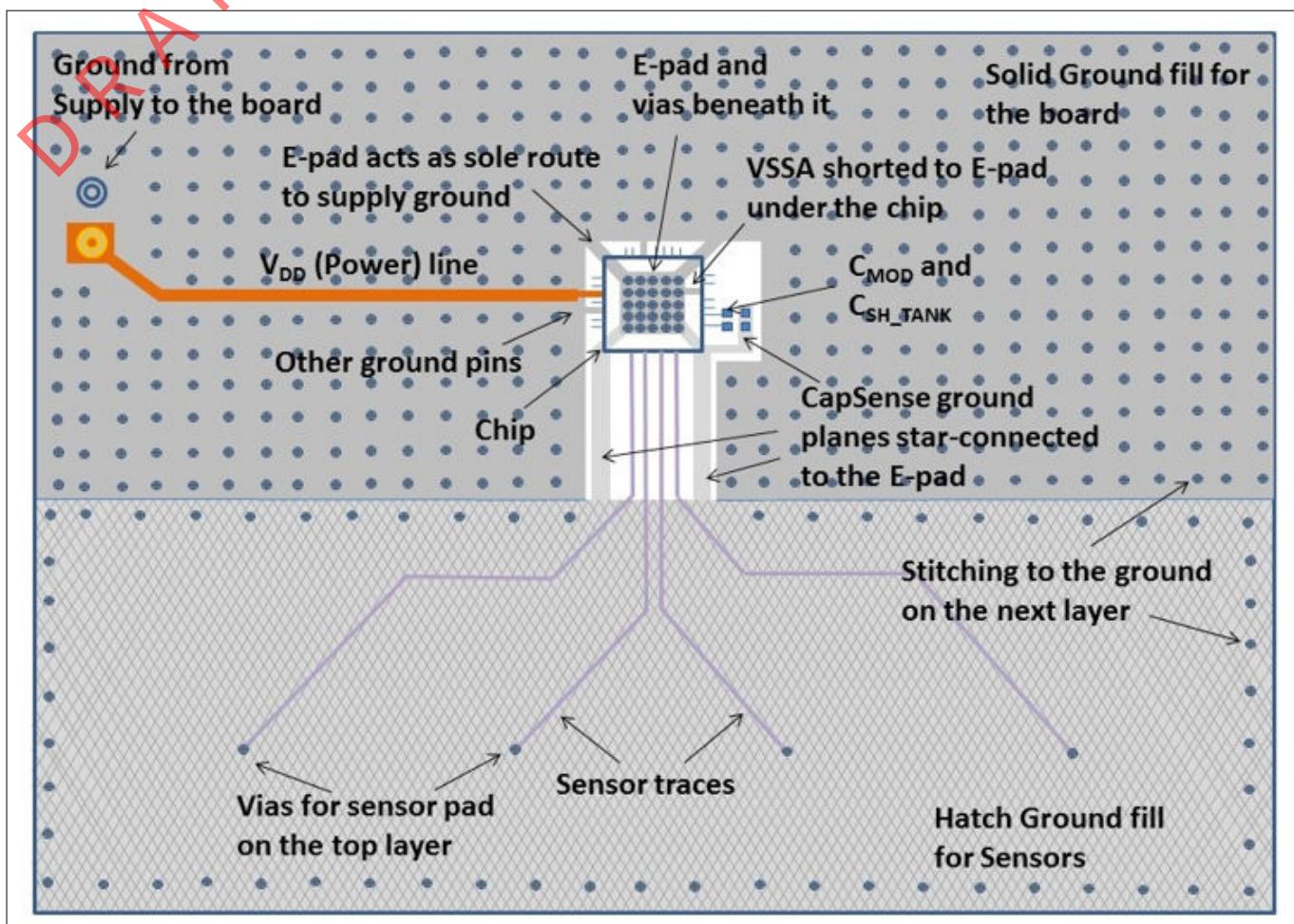


Figure 337 PCB bottom layer layout with PSoC™ 4 Bluetooth® LE (with E-pad)

5.8.7.4.11 Power supply layout recommendations

CAPSENSE™ is a high-sensitivity analog system. Therefore, a poor PCB layout introduces noise in high-sensitivity sensor configurations such as proximity sensors and buttons with thick overlays (>1 mm). To achieve low noise in a high-sensitivity CAPSENSE™ design, the PCB layout should have decoupling capacitors on the power lines, as listed in [Table 67](#).

Table 67 Decoupling capacitors on power lines

Power line	Decoupling capacitors	Corresponding ground terminal	Applicable device family
VDD	0.1 µF and 1 µF	VSS	PSoC™ 4000
VDDIO	0.1 µF and 1 µF	VSS	PSoC™ 4000, PSoC™ 6 MCU
VDDD	0.1 µF and 1 µF	VSS	PSoC™ 4100, PSoC™ 4200, PSoC™ 6 MCU
	0.1 µF and 1 µF	VSSD	PSoC™ 4100-BL, PSoC™ 4200-BL, PSoC™ 4200L, PSoC™ S-series, PSoC™ 4100S Plus, PSoC™ 4100S Max

(table continues...)

5 PSoC™ 6 application notes

Table 67 (continued) Decoupling capacitors on power lines

Power line	Decoupling capacitors	Corresponding ground terminal	Applicable device family
VDDA ²⁷⁾	0.1 µF and 1 µF (Battery powered supply)	VSSA	PSoC™ 4100, PSoC™ 4200, PSoC™ 4100-BL, PSoC™ 4200-BL, PSoC™ 4200L, PSoC™ 4S-Series, PSoC™ 4100S Plus, PSoC™ 4100PS, PSoC™ 6 MCU
	0.1 µF and 10 µF (Mains Powered supply)	VSSA	PSoC™ 4S-series, PSoC™ 4100S Plus, PSoC™ 4100PS
VDDR	0.1 µF and 1 µF	VSSD	PSoC™ 4100-BL, PSoC™ 4200-BL, PSoC™ 6 MCU with Bluetooth® LE Connectivity
VCCD	See device datasheet	VSS (PSoC™ 4000) or VSSD (all others)	All device families

The decoupling capacitors and C_{MOD} capacitor must be placed as close to the chip as possible to keep ground impedance and supply trace length as low as possible.

For further details on bypass capacitors, see the Power section in the [Device datasheet](#).

5.8.7.4.12 Layout guidelines for liquid tolerance

As explained in the [Liquid tolerance](#) section, by implementing a shield electrode and a guard sensor, a liquid-tolerant™ system can be implemented. If there are multiple CSD blocks in the device, each CSD block should have a dedicated shield electrode. This section shows how to implement a shield electrode and a guard sensor.

Layout guidelines for shield electrode

The area of the shield electrode depends on the size of the liquid droplet and the area available on the board for implementing the shield electrode. The shield electrode should surround the sensor pads and traces, and spread no further than 1 cm from these features. Spreading the shield electrode beyond 1 cm has negligible effect on system performance.

Also, having a large shield electrode may increase radiated emissions. If the board area is very large, the area outside the 1-cm shield electrode should be left empty, as [Figure 338](#) shows. The board design should focus on reducing the coupling capacitance between the liquid droplet and ground. Thus, for improved liquid tolerance, there should not be any hatch fill or a trace connected to ground in the top and bottom layers of the PCB.

When there is a grounded hatch fill or a trace then, when a liquid droplet falls on the touch surface, it may cause sensor false triggers. Even if there is a shield electrode between the sensor and ground, the effect of the shield electrode will be totally masked out and sensors may false trigger.

In some applications, there may not be sufficient area available on the PCB for shield electrode implementation. In such cases, the shield electrode can spread less than 1 cm; the minimum area for shield electrode can be the area remaining on the board after implementing the sensor.

²⁷ The V_{DDA} pin on PSoC™ 4 S-Series, PSoC™ 4100S Plus, and PSoC™ 4100PS family requires different values of bulk capacitor depending on the power supply source. If the device is battery powered, it is recommended to use 0.1 µF and 1 µF capacitors in parallel and if the device is mains powered, it is recommended to use 0.1 µF and 10 µF in parallel. This is to improve the power supply rejection ratio of reference generator (REFGEN) used in the CAPSENSE™ block.

~~5 PSoC™ 6 application notes~~

In some applications, the capacitance of the shield electrode will be very high; you can reduce it with the following techniques:

- Using multiple shield electrode instead of single shield electrode: If there is a single hatch pattern with a higher C_p , split the hatch pattern into multiple hatch patterns and drive it with the shield signal to decrease the shield C_p . This will also allow the use of a higher range of sense clock frequencies for the sensors which will improve the sensitivity of the CAPSENSE™ system. In a complex layout design, this approach will make trace routing simple
- Connecting multiple shield pins to the same electrode: If splitting the shield electrode in the layout is not feasible, connect multiple shield pins to the same electrode. This will make all the series resistance of the sensor pins in parallel and reduce the effective time constant of the shield electrode, which will allow using a higher range of sense clock frequencies for sensors, which will improve the sensitivity of the CAPSENSE™ system

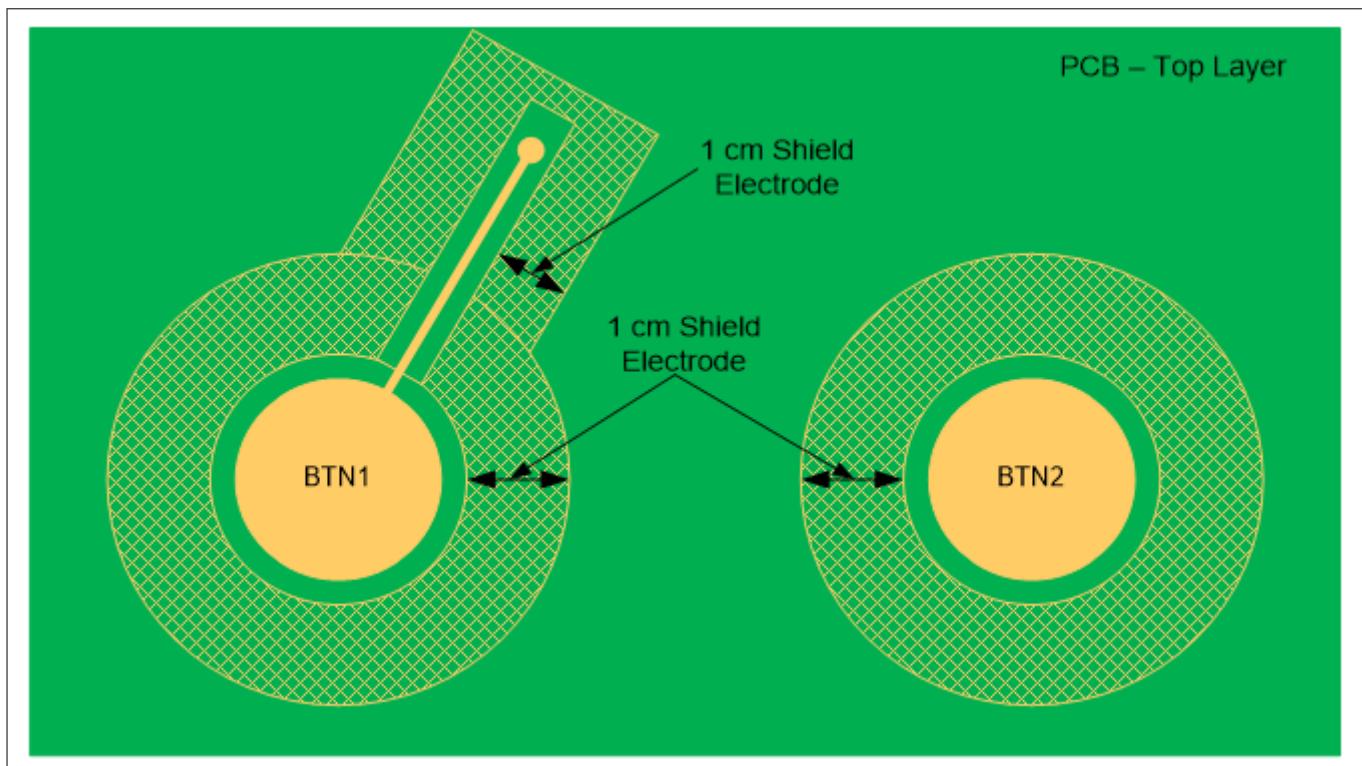


Figure 338 Shield electrode placement when sensor trace Is routed in top and bottom layer

Follow these guidelines to implement the shield electrode in two-layer and four-layer PCBs:

Two-layer PCB:

- Top layer: Hatch fill with 7 mil trace and 45 mil grid (25 percent fill). Hatch fill should be connected to the driven-shield signal
- Bottom layer: Hatch fill with 7 mil trace and 70 mil grid (17 percent fill). Hatch fill should be connected to the driven-shield signal

Four (or more)-layer PCB:

- Top layer: Hatch fill with 7 mil trace and 45 mil grid (25 percent fill). Hatch fill should be connected to the driven-shield signal
- Layer-2: Hatch fill with 7 mil trace and 70 mil grid (17 percent fill). Hatch fill should be connected to the driven-shield signal
- Layer-3: V_{DD} Plane
- Bottom layer: Hatch fill with 7 mil trace and 70 mil grid (17 percent fill). Hatch fill should be connected to ground

~~5 PSoC™ 6 application notes~~

The recommended air gap between the sensor and the shield electrode is 1 mm.

Layout guidelines for guard sensor

As explained in the [Guard sensor](#) section, the guard sensor is a copper trace that surrounds all sensors, as Figure 339 shows.

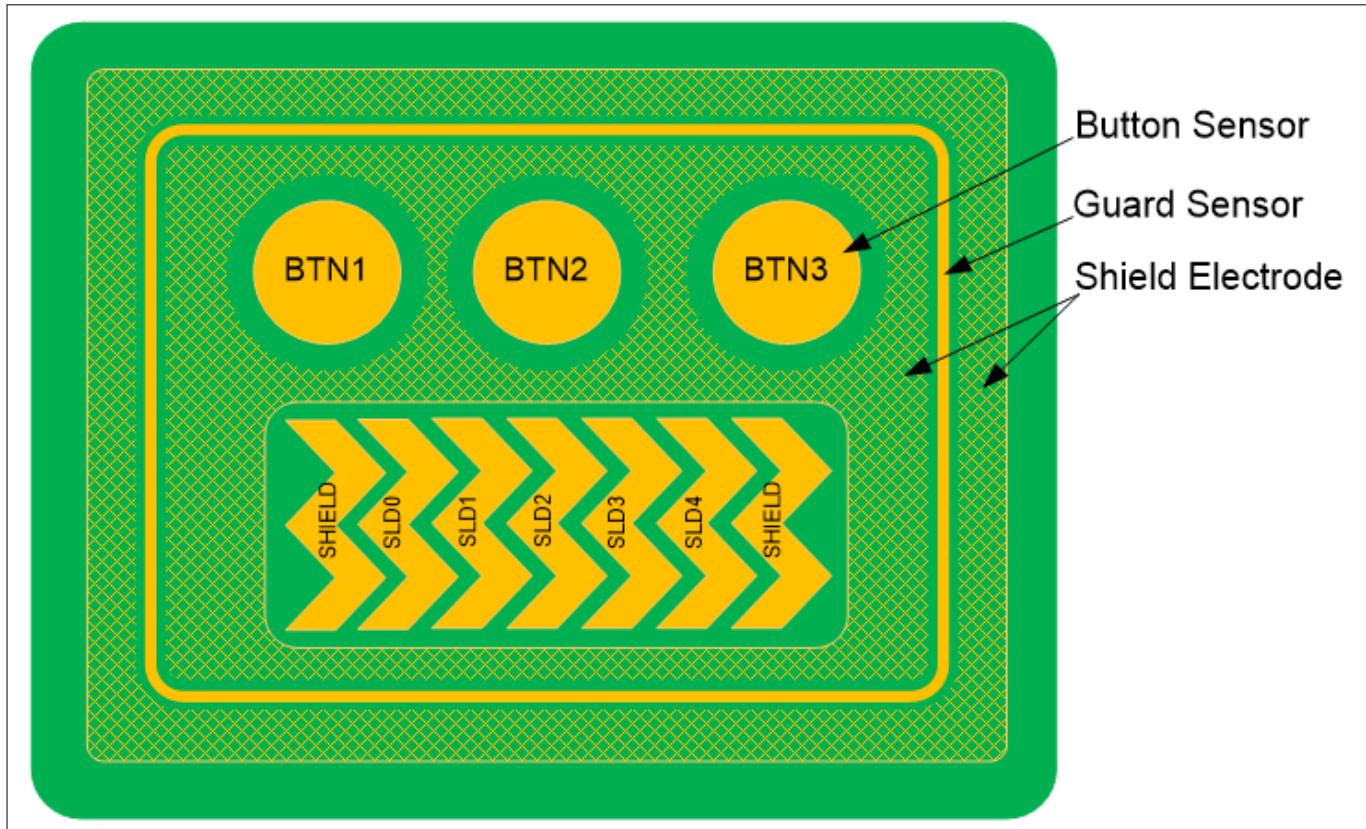


Figure 339 PCB layout with shield electrode and guard sensor

The guard sensor should be triggered only when there is a liquid stream on the touch surface. Ensure that the shield electrode pattern surrounds the guard sensor to prevent it from turning on due to liquid droplets. The guard sensor should be placed such that it meets the following conditions:

- It should be the first sensor to turn on when there is a liquid stream on the touch surface. To accomplish this, the guard sensor is usually placed such that it surrounds all sensors
- It should not be accidentally touched while pressing a button or slider sensor. Otherwise, the button sensors and slider sensor scanning will be disabled and the CAPSENSE™ system will become nonoperational until the guard sensor is turned off. To ensure the guard sensor is not accidentally triggered, place the guard sensor at a distance greater than 1 cm from the sensors

Follow these guidelines to implement the guard sensor:

- The guard sensor should be in the shape of a rectangle with curved edges and should surround all the sensors
- The recommended thickness for a guard sensor is 2 mm
- The recommended clearance between the guard sensor and the shield electrode is 1 mm

If there is no space on the PCB for implementing a guard sensor, the guard sensor functionality can be implemented in the firmware. For example, you can use the ON/OFF status of different sensors to detect a liquid stream depending on the use case data.

The following conditions can be used to detect a liquid stream on the touch surface:

~~5 PSoC™ 6 application notes~~

- When there is a liquid stream, more than one button sensor will be active at a time. If your design does not require multi-touch sensing, you can detect this and reject the sensor status of all the button sensors to prevent false triggering
- In a slider, if the slider segments which are turned ON are not adjacent segments, you can reset the slider segments status or reject the slider centroid value that is calculated
- A firmware algorithm to detect the false touch due to water drop from the use case data can be made to improve the false touch rejection capability sensors

Liquid tolerance with ground ring

In some applications, it is required to have a ground ring (solid trace or a hatch fill) around the periphery of the board for improved ESD and EMI/EMC performance, as shown in [Figure 340](#). Having a ground ring around the board may result in sensor false triggers when liquid droplets fall in between the sensor and the ground sensor. Therefore, it is recommended not to have any ground in the top layer. If the design must have a ground ring in the top layer, use a ground ring with the minimum thickness (8 mils).

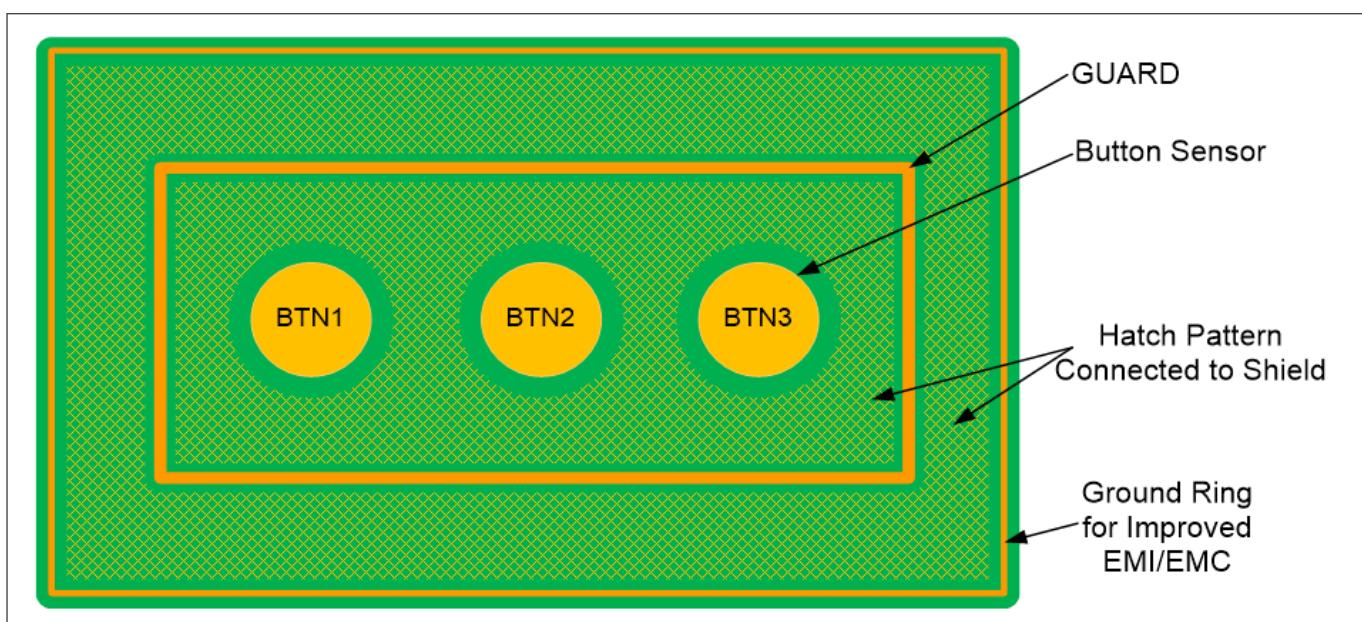


Figure 340 CAPSENSE™ design with ground ring for improved ESD and EMI/EMC performance

5.8.7.4.13 Schematic rule checklist

[Table 68](#) provides the checklist to verify your CAPSENSE™ schematic.

Table 68 Schematic Rule Checklist

No.	Category	Recommendations/Remarks
1	C_{MOD}	2.2 nF. See Table 69 for pin selection.
2	C_{SH_TANK}	10 nF if shield electrode is being used, NA otherwise. See Driven shield signal and shield electrode and CAPSENSE™ CSD shielding for details on shield electrode and use of C_{SH_TANK} respectively. See Table 69 for pin selection.
3	C_{INTA}/C_{INTB}	470 pF. See Table 69 for pin selection.
3	Series resistance on input lines	560 Ω for Self-capacitance and 2 k Ω for mutual-capacitance. See Series resistors on CAPSENSE pins for details.

(table continues...)

Reference manual

5 PSoC™ 6 application notes

~~DRAFT~~ **Table 68 (continued) Schematic Rule Checklist**

No.	Category	Recommendations/Remarks
4	Sensor pin selection	If possible, avoid pins that are close to the GPIOs carrying switching/communication signals. Physically separate DC loads such as LEDs and I ² C pins from the CAPSENSE™ pins by a full port wherever possible. See Sensor pin selection section for more details.
5	GPIO Source/Sink Current	Ensure that the total sink current through GPIOs is not greater than 40 mA when the CAPSENSE™ block is scanning the sensors.

External capacitors pin selection

As explained in the [CAPSENSE™ fundamentals](#) section, CAPSENSE™ require external capacitors - C_{MOD} (CSD sensing method), C_{TANK} (Only when Shield is implemented), and C_{INTX} (CSX sensing method) for reliable operation. Starting from [PSoC™ Creator 3.3 SP2](#), the number of pins that can support C_{MOD} and C_{SH_TANK} is increased to improve design flexibility. [Table 69](#) lists the recommended pins for C_{MOD} , C_{INTX} and C_{SH_TANK} capacitors for PSoC™ Creator 3.3 SP2 or later versions.

Note: *For PSoC™ 4100/PSoC™ 4200, if a pin other than P4[2] is selected for C_{MOD} , P4[2] will not be available for any other function. For example, if you try routing C_{MOD} to P2[0] in PSoC™ Creator for a PSoC™ 4200 device, it uses both P2[0] and P4[2].*

Table 69 Recommended pins for external capacitors

Device	C_{MOD} (or C_{MOD1} for Fifth-Generation CAPSENSE™)	C_{SH_TANK} (or C_{MOD2} for Fifth-Generation CAPSENSE™)
PSoC™ 4000	P0[4]	P0[2]
PSoC™ 4100/PSoC™ 4200	P4[2]	P4[3]
PSoC™ 4200M/PSoC™ 4200L	CSD0: P4[2]	CSD0: P4[3]
	CSD1: P5[0]	CSD1: P5[1]
PSoC™ 4 Bluetooth® LE	P4[0]	P4[1]
PSoC™ 6 MCU	P7[1]	P7[2]
PSoC™ 4S-Series, PSoC™ 4100S Plus	P4[2]	P4[3]
PSoC™ 4100PS	P5[2]	P5[3]
PSoC™ 4100S Max	Channel0: P4[0]	Channel0: P4[1]
	Channel1: P7[0]	Channel1: P7[1]

Table 70 Supported pins for external capacitors

Device	C_{MOD} (or C_{MOD1} for fifth-generation CAPSENSE™)	C_{SH_TANK} (or C_{MOD2} for fifth-generation CAPSENSE™)	C_{INTA}	C_{INTB}
PSoC™ 4000	Port0[0:7], Port1[0:7]P2[0]	Port0 [0:7], Port1 [0:7]P2[0]	P0[4]	P0[2]
PSoC™ 4100	Port0 [0:7], Port1 [0:7]Port2 [0:7], Port3 [0:7]P4[2]	Port0 [0:7], Port1 [0:7]Port2 [0:7], Port3 [0:7]P4[3]	Not supported	Not supported

(table continues...)

5 PSoC™ 6 application notes

Table 70 (continued) Supported pins for external capacitors

Device	C_{MOD} (or C_{MOD1} for fifth-generation CAPSENSE™)	C_{SH_TANK} (or C_{MOD2} for fifth-generation CAPSENSE™)	C_{INTA}	$C_{INT\ B}$
PSoC™ 4200	Port0 [0:7], Port1 [0:7]Port2 [0:7], Port3 [0:7]P4[2]	Port0 [0:7], Port1 [0:7], Port2 [0:7], Port3 [0:7]P4[3]	Port0 [0:7], Port1 [0:7]Port2 [0:7], Port3 [0:7]	Port0 [0:7], Port1 [0:7]Port2 [0:7], Port3 [0:7]
PSoC™ 4200M	CSD0: Port0 [0:7], Port1 [0:7] Port2 [0:7], Port3 [0:7] Port4 [0:6], Port6 [0:5] Port7 [0:1]	CSD0: Port0 [0:7], Port1 [0:7]Port2 [0:7], Port3 [0:7] Port4 [0:6], Port6 [0:5] Port7 [0:1]	CSD0: P4[2]	CSD0: P4[3]
	CSD1: Not supported	CSD1: Not supported	CSD1: Not supported	CSD1: Not supported
PSoC™ 4200L	CSD0: Port0 [0:7], Port1 [0:7] Port2 [0:7], Port3 [0:7] Port4 [0:6], Port6 [0:5] Port7 [0:7], Port10 [0:7], Port11 [0:7]	CSD0: Port0 [0:7], Port1 [0:7] Port2 [0:7], Port3 [0:7] Port4 [0:6], Port6 [0:5] Port7 [0:7], Port10 [0:7] Port11 [0:7]	CSD0: P4[2]	CSD0: P4[3]
	CSD1: Port5 [0:7], Port8 [0:7] Port9 [0:7]	CSD1: Port5 [0:7], Port8 [0:7] Port9 [0:7]	CSD1: P5[0]	CSD1: P5[1]
PSoC™ 4 Bluetooth® LE	Port0 [0:7], Port1 [0:7]Port2 [0:7], Port3 [0:7]Port4 [0:1], Port5 [0:1]Port6 [0:1]	Port0 [0:7], Port1 [0:7]Port2 [0:7], Port3 [0:7]Port4 [0:1], Port5 [0:1]Port6 [0:1]	P4[0]	P4[1]
PSoC™ 6 MCU	P7[1] or P7[2] or P7[7]	P7[1] or P7[2] or P7[7]	P7[1]	P7[2]
PSoC™ 4S-Series, PSoC™ 4100S Plus	P4[2], P4[3], P4[1]	P4[2], P4[3], P4[1]	P4[2]	P4[3]
PSoC™ 4100PS	P5[0], P5[2], P5[3]	P5[0], P5[2], P5[3]	P5[2]	P5[3]
PSoC™ 4100S Max	Channel0: P4[0], P4[2]	Channel0: P4[1], P4[3]	Not applicable	Not applicable
	Channel1: P7[0], P5[1]	Channel1: P7[1], P5[2]		

Sensor pin selection

As explained in [CAPSENSE™ fundamentals](#), PSoC™ supports CSD and CSX capacitive sensing methods. Each CSD sensor requires a single sensor pin and CSX sensor will require two sensor pins for Tx and Rx electrode in addition to the required external capacitors for each sensing technique.

The selection of the sensor pins should be in a way such that the CAPSENSE™ sensor traces and communication or other toggling GPIO traces are isolated by proper port/pin assignment. The following are some recommended guidelines:

~~5 PSoC™ 6 application notes~~

- Isolate switching signals, such as PWM, I2C communication lines, and LEDs from the sensor and sensor traces. Place them at least 4 mm apart and fill a hatched ground between the CAPSENSE™ traces and the switching signals to avoid crosstalk
- Distribute the placement of DC loads on different ports to reduce the noise in CAPSENSE™. It is recommended to have digital I/Os spread on different ports rather than concentrating in a single port
- While the CAPSENSE™ block is scanning the sensor, limit the total source or sink current through GPIOs to less than 40 mA while the CAPSENSE™ block is scanning the sensor. Sinking a current greater than 40 mA while the CAPSENSE™ sensor is scanning may result in excessive noise in the sensor raw count
- For a PSoC™ 4 device it is recommended to place all the digital DC loads like LEDs, I2C/UART communication pins on the port powered by only VSSD; see the [Device datasheet](#) for determining the ports that are powered by VSSD. Placing DC loads on ports powered by VSSA will shift the VSSA up. Since CAPSENSE™ is powered by VSSA, it will affect its performance
- For PSoC™ 6 family of devices:
 - [Table 71](#) lists the ports that support CAPSENSE™, selecting ports 5, 6, 7, and 8 for CAPSENSE™ ensures lesser noise
 - It is recommended to place all digital switching pins such as LEDs, I²C, UART, SPI, SMIF communication pins on the ports that are powered by a different power supply domain which is not shared with the CAPSENSE™ ports. [Table 72](#) lists the ports, their supply domains, and recommendations for using these ports with CAPSENSE™. For more details, see the Errata section of the [Device datasheet](#). A deviation from these guidelines might cause a noise due to level shift in raw count. For more details, see [Raw counts show a level-shift or increased noise when GPIOs are toggled](#). To isolate the supply domains further, it is better to externally isolate them using ferrite beads as shown in [Figure 342](#)

Table 71 CAPSENSE™ capable ports in PSoC™ 6 devices

Device	CAPSENSE™ capable ports
CY8C62x6, CY8C62x7	P0, P1, P2, P4, P5, P6, P7, P8, P9, P10, P11
CY8C63x6, CY8C63x7	P0, P1, P2, P4, P5, P6, P7, P8, P9, P10, P11
CY8C62x5	P7.0 to P7.7, P8.0 to P8.3, P9.0 to P9.3

Table 72 Recommendations of port usage with CAPSENSE™ for PSoC™ 6 device

Ports	Supply domain	Recommended for CAPSENSE™	Recommendations for GPIOs if used for communication, LEDs, and other high frequency functionality with CAPSENSE™
P0	VBACKUP	No*	Switching frequency < 8 MHz
P1	VDDD	No*	Switching frequency < 1 MHz, SLOW Slew Rate
P2, P3, P4	VDDIO2	No*	Switching frequency < 25 MHz
P5, P6, P7, P8	VDDIO1	Yes	Not recommended
P9, P10	VDDIOA	No*	Switching frequency < 1 MHz, SLOW Slew Rate
P11, P12, P13	VDDIO0	No*	Switching frequency < 80 MHz
P14	VDDUSB	No*	NA

Note:

* If you need additional CAPSENSE™ pins and if you must use GPIOs in ports P1, P9, and P10 as Tx electrode for CSX sensor, restrict the Tx clock frequency within 1 MHz and use SLOW slew rate. [Figure](#)

5 PSoC™ 6 application notes

~~DRAFT~~

341 shows an example on how to select the **Slew Rate** of the GPIO using the **Device configurator** in the **ModusToolbox™** project. Note that using the ports other than the recommended ports for **CAPSENSE™** might cause higher noise in raw count.

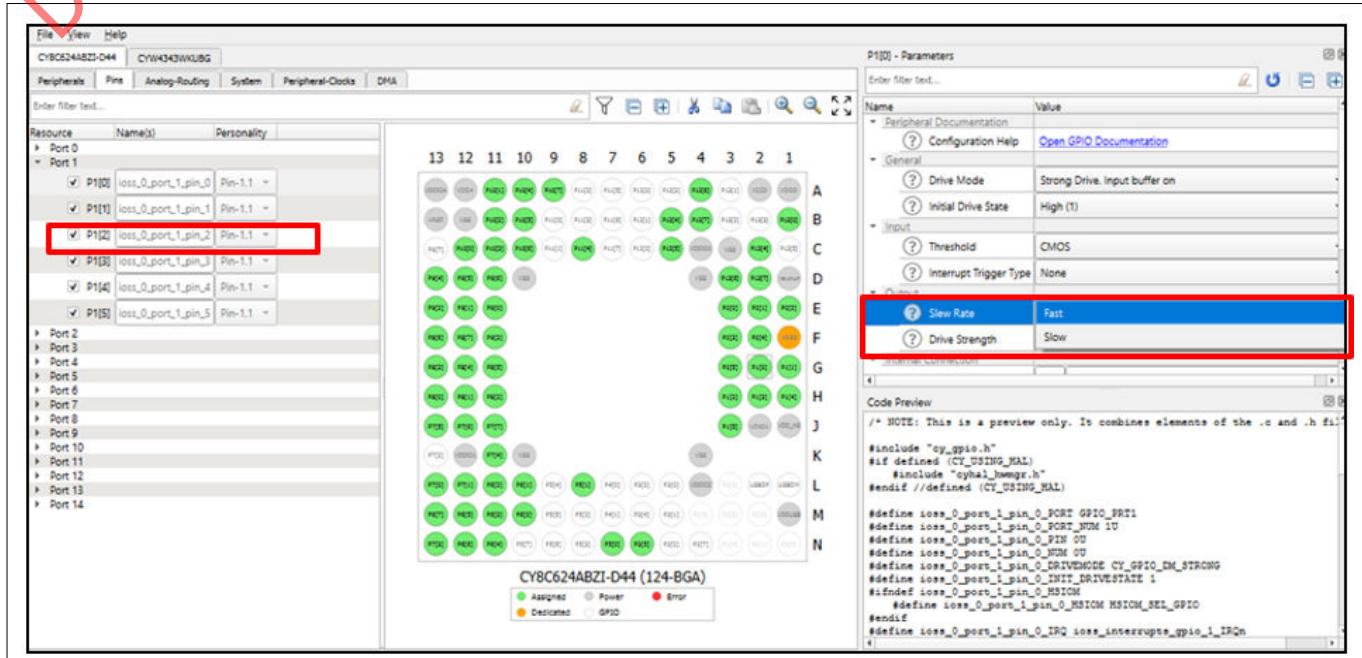


Figure 341 Selecting slew rate for GPIOs

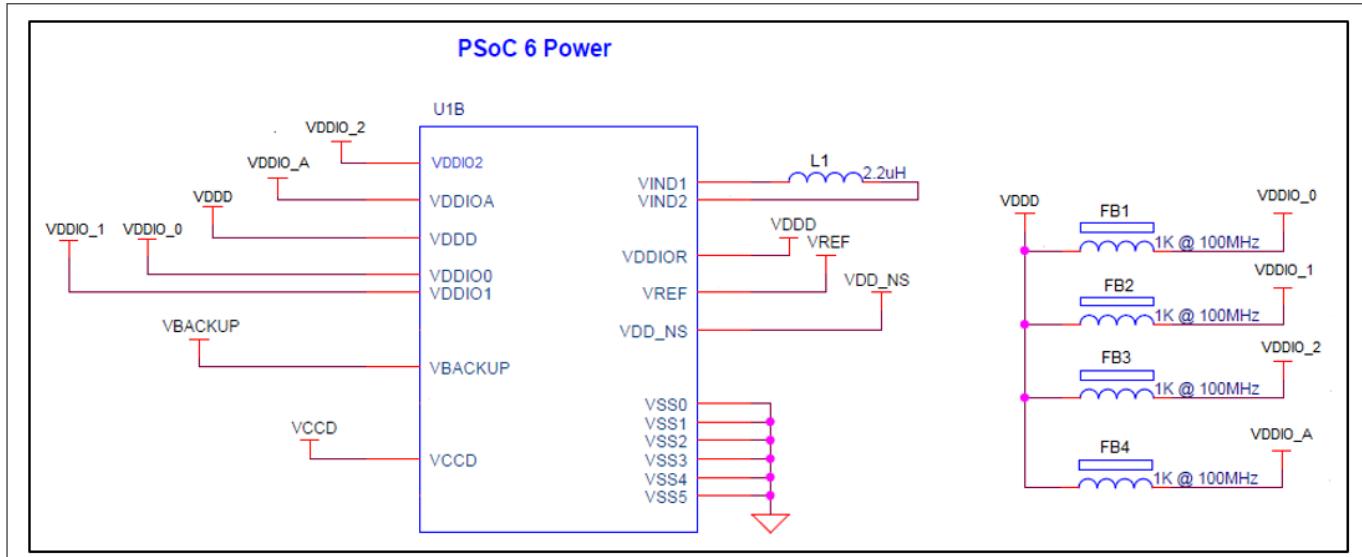


Figure 342 Externally isolated supply domains

5.8.7.4.14 Layout rule checklist

Table 73 provides the checklist to help verify your layout design.

~~5 PSoC™ 6 application notes~~

Table 73 Layout rule checklist

No.	Category		Minimum value	Maximum value	Recommendations/Remarks
1	Button	Shape	N/A	N/A	Circle or rectangular with curved edges
		Size	5 mm	15 mm	10 mm
		Clearance to ground hatch	0.5 mm	2 mm	Should be equal to overlay thickness
2	Slider	Width of segment	1.5 mm	8 mm	8 mm
		Clearance between segments	0.5 mm	2 mm	0.5 mm
		Height of segment	7 mm	15 mm	12 mm
3	Overlay	Type	N/A	N/A	Material with high relative permittivity (except conductors) Remove any air gap between sensor board and overlay/front panel of the casing.
		Thickness for buttons	N/A	5 mm	
		Thickness for sliders	N/A	5 mm	
		Thickness for touchpads	N/A	0.5 mm	
4	Sensor traces	Width	N/A	7 mil	Use the minimum width possible with the PCB technology that you use.
		Length	N/A	300 mm for a standard (FR4) PCB 50 mm for flex PCB	Keep as low as possible.
		Clearance to ground and other traces	0.25 mm	N/A	Use maximum clearance while keeping the trace length as low as possible.
		Routing	N/A	N/A	Route on the opposite side of the sensor layer. Isolate from other traces. If any non- CAPSENSE™ trace crosses the CAPSENSE™ trace, ensure that intersection is orthogonal. Do not use sharp turns.

(table continues...)

~~DRAFT~~
5 PSoC™ 6 application notes

Table 73 (continued) Layout rule checklist

No.	Category		Minimum value	Maximum value	Recommendations/Remarks
5	Via	Number of vias	1	2	At least one via is required to route the traces on the opposite side of the sensor layer.
		Hole size	N/A	N/A	10 mil
6	Ground	Hatch fill percentage	N/A	N/A	Use hatch ground to reduce parasitic capacitance. Typical hatching: 25% on the top layer (7 mil line, 45 mil spacing) 17% on the bottom layer (7 mil line, 70 mil spacing)
7	Series resistor	Placement	N/A	N/A	Place the resistor within 10 mm of the PSoC™ pin. See Figure 343 for an example placement of series resistance on board.
8	Shield electrode	Spread	N/A	1 cm	If you have PCB space, use 1 cm spread.
9	Guard sensor (for water tolerance)	Shape	N/A	N/A	Rectangle with curved edges
		Thickness	N/A	N/A	Recommended thickness of guard trace is 2 mm and distance of guard trace to shield electrode is 1 mm.
10	C_{MOD}	Placement	N/A	N/A	Place close to the PSoC™ pin. See Figure 343 for an example placement of C_{MOD} on PCB.
11	C_{SH_TANK}	Placement	N/A	N/A	Place close to the PSoC™ pin. See Figure 343 for an example placement of C_{SH_TANK} on board.
12	C_{INTA}	Placement	N/A	N/A	Place close to the PSoC™ pin. See Figure 343 for an example placement of C_{INTA} on the PCB.
13	C_{INTB}	Placement	N/A	N/A	Place close to the PSoC™ pin. See Figure 343 for an example placement of C_{INTB} on the PCB.

5 PSoC™ 6 application notes

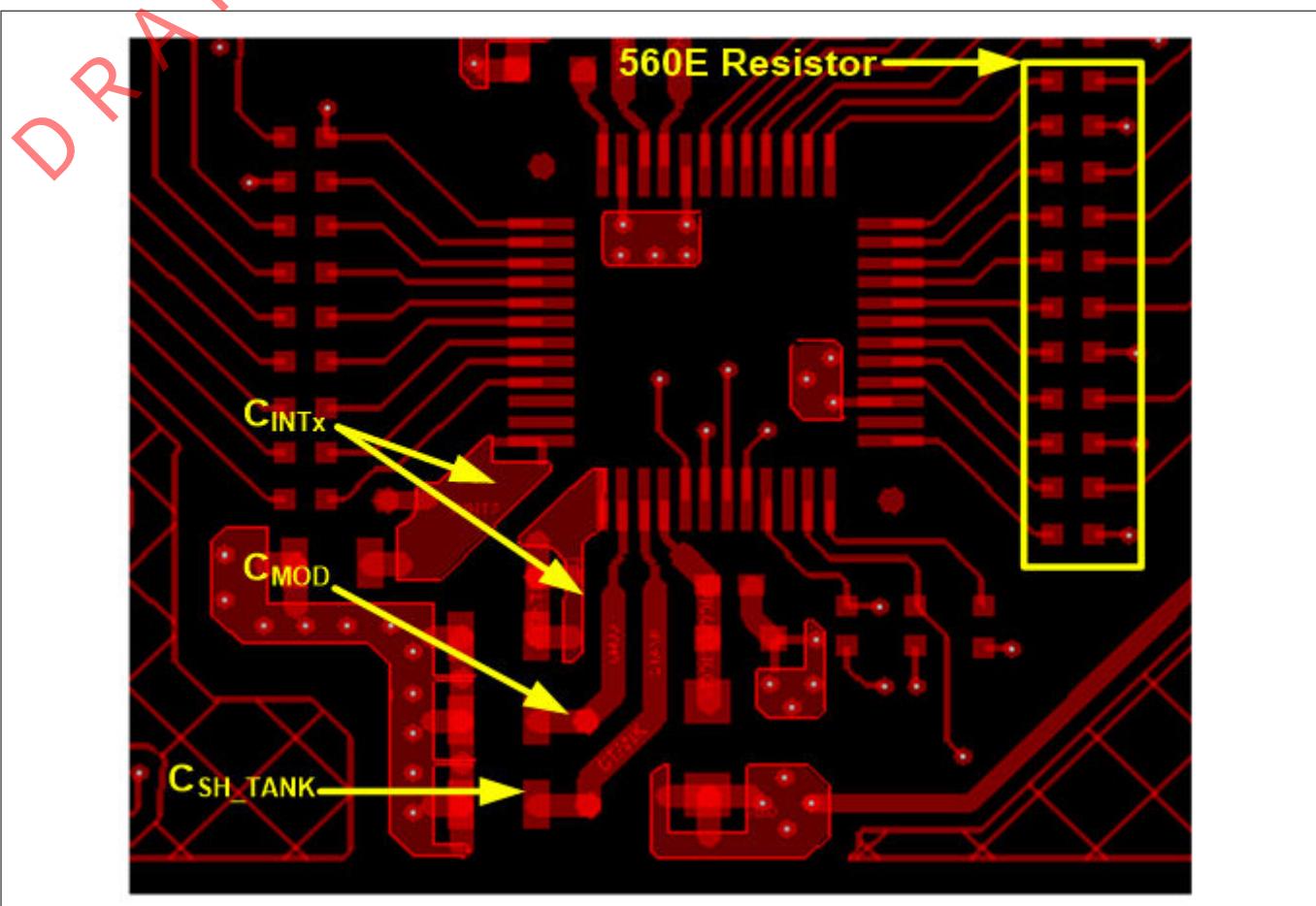


Figure 343 Example placement for C_{MOD} , C_{INTx} , C_{SH_TANK} , and series resistance on input lines in PSoC™ 4200M device

5.8.7.5 Noise in CAPSENSE™ system

5.8.7.5.1 Finger injected noise

If the power supply design of the system is poor, the power and ground supplies of a device fluctuates in voltage relative to the finger ground (earth ground) in a common mode fashion. This type of noise is called common mode noise. [Figure 344](#) illustrates the common mode noise, where both the 5 V and the 0 V output leads of the power supply remain 5 V from each other, but they move up and down together, in a “common mode” manner.

This is not a problem, until a finger touch occurs on the button. A finger touch on the button introduces a (capacitive) path to the same earth ground and it will create a path for charge flow, which is equivalent to a noise signal injected exactly at the finger touch location. This injected noise caused by the common mode noise in power supply is called finger injected noise. It is observed only during the finger touch on the button in AC powered application and it doesn't occur in battery powered application.

5 PSoC™ 6 application notes

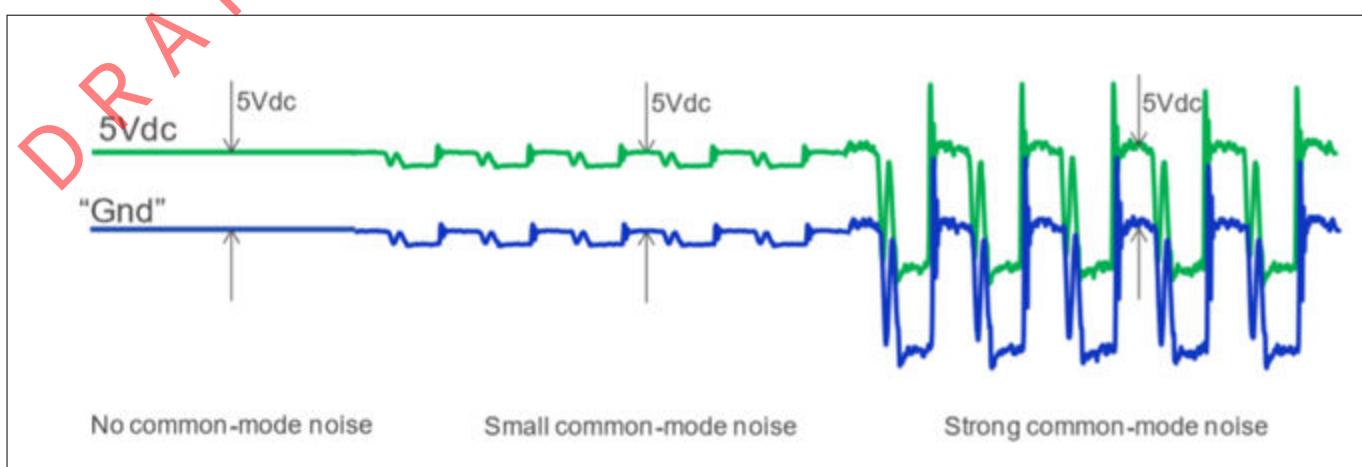


Figure 344 Common mode noise in the power supply

Note: *When the complete system powered by AC supply is held in hand of the user, the entire system will be grounded to earth sufficiently and no significant “common-mode” noise would flow through the touching finger to earth. However, if the system is connected to the power supply and placed on a desk, a touch on the button, can introduce a problematic discharge path to ground.*

Recommendations to reduce the finger injected noise

The finger injected noise could be reduced by properly following the layout and schematics guidelines described in this section. The general recommendations to reduce the finger injected noise is explained below.

1. Fill the PCB board around the button with hatched pattern and connected it to device ground. Follow the recommendations as mentioned in the section [Ground plane](#).

[Figure 345](#) shows the impact of ground on the finger injected noise for mutual capacitance button and it is also true for CSD sensing technique. In the left figure, the system doesn't have the hatched ground around the button and most of the injected noise through the finger pass to the Rx pin of the device through the Capacitance formed between the finger and Rx electrode. In the right figure, the system has the hatched ground around the button and thus the finger injected noise is having an alternate path to flow which results in the reduction of the noise reaching to the device Rx pin.

5 PSoC™ 6 application notes

DRAFT

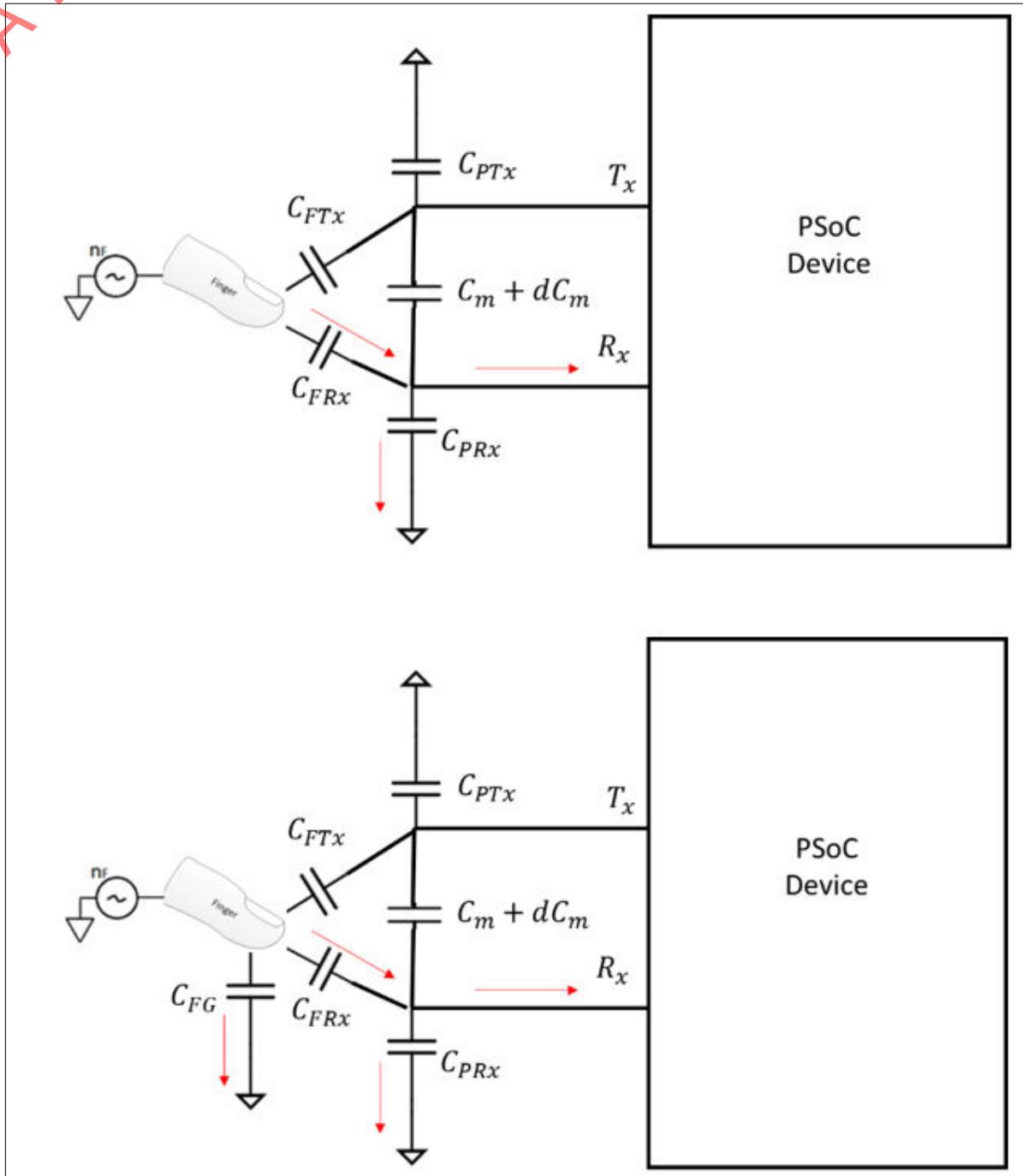


Figure 345 Effect of ground on finger injected noise

2. Better power supply design of the system could easily eliminate the common mode noise, which will in turn reduce the finger injected noise
3. Use software technique that are available in the CAPSENSE™ component to combat the finger injected noise such as selecting optimal sensing clock frequency and Multi frequency scanning, and so on
4. Increase the overlay thickness will reduce the finger injected noise as it will decrease the capacitance formed between the finger and Rx electrode

5.8.7.5.2 VDDA noise

The noise in the system due to unwanted voltage ripples in the VDD supply is called VDDA noise.

~~5 PSoC™ 6 application notes~~

~~Recommendations to reduce the VDDA noise~~

The VDDA noise could be reduced by properly following the layout and schematics guidelines in this chapter. The general recommendations to reduce the VDDA noise as follows:

1. Use clean power supply and have VDD ripples below the limits mentioned in the device datasheet
2. Use filters or LDO regulator in the VDD power lines
3. Use decoupling capacitors on the power supply pins to reduce the conducted noise from the power supply
4. To reduce high-frequency noise, place a ferrite bead around power supply or communication lines
5. Selecting the proper supply configuration as mentioned in the Power section in the [Device datasheet](#) and using the internal regulator to the device might help in reducing the VDDA noise

5.8.7.5.3 External noise

Any noise that is injected into to the system through the routing trace lines like ESD, EMI, conducted noise are coming into the category of external noise. The recommended guidelines for reducing the impact of the external noise are discussed in this section.

ESD protection

The nonconductive overlay material used in CAPSENSE™ provides inherent protection against ESD. [Table 74](#) lists the thickness of various overlay materials, required to protect the CAPSENSE™ sensors from a 12 kV discharge (according to the IEC 61000 4 2 specification).

Table 74 Overlay thickness for ESD protection

Material	Breakdown voltage (V/mm)	Minimum overlay thickness for protection against 12 kV ESD (mm)
Air	1200 – 2800	10
Wood – dry	3900	3
Glass – common	7900	1.5
Glass – Borosilicate (Pyrex®)	13,000	0.9
PMMA Plastic (Plexiglas®)	13,000	0.9
ABS	16,000	0.8
Polycarbonate (Lexan®)	16,000	0.8
Formica	18,000	0.7
FR-4	28,000	0.4
PET Film (Mylar®)	280,000	0.04
Polyimide Film (Kapton®)	290,000	0.04

If the overlay material does not provide sufficient protection (for example, ESD from other directions), you can apply other ESD counter-measures, in the following order: [Prevent](#), [Redirect](#), and [ESD protection devices](#).

Preventing ESD discharge

Preventing the ESD discharge from reaching the PSoC™ is the best countermeasure you can take. Make sure that all paths to PSoC™ have a breakdown voltage greater than the maximum ESD voltage possible at the surface of the equipment. You should also maintain an appropriate distance between the PSoC™ and possible ESD

5 PSoC™ 6 application notes

DRAFT

sources. In the example illustrated in [Figure 346](#), if L1 and L2 are greater than 10 mm, the system can withstand a 12 kV ESD.

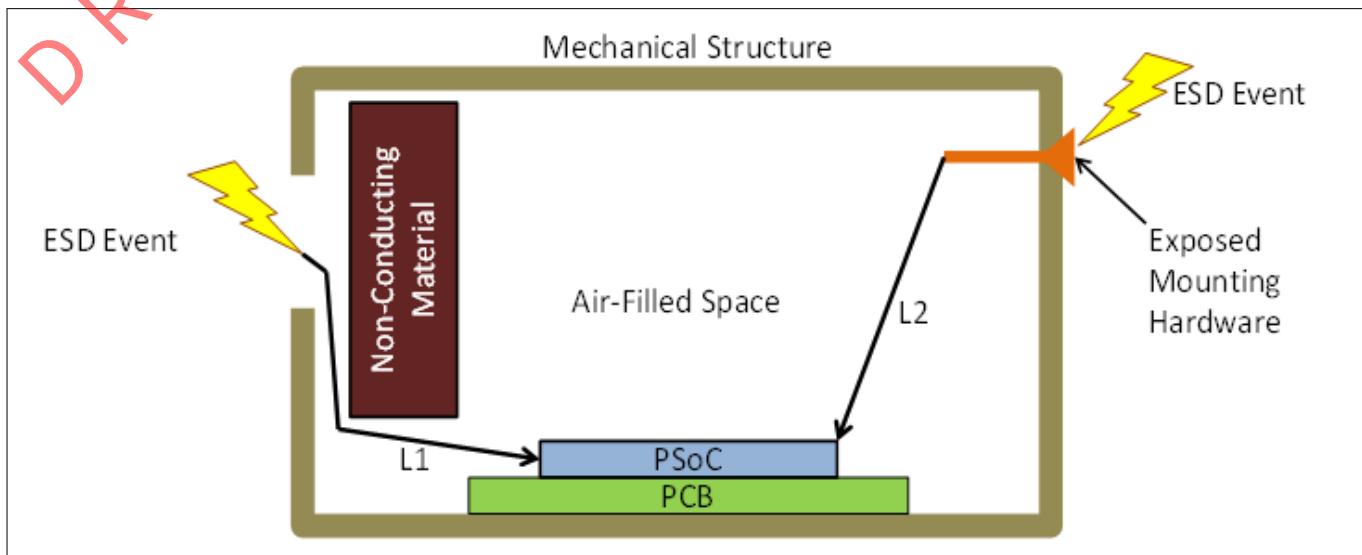


Figure 346 **ESD paths**

If it is not possible to maintain adequate distance, place a protective layer of nonconductive material with a high breakdown voltage between the possible ESD source and PSoC™. One layer of 5 mil thick Kapton® tape can withstand 18 kV. See [Table 74](#) for other material dielectric strengths.

Redirect

If your product is densely packed, preventing the discharge event may not be possible. In such cases, you can protect the PSoC™ from ESD by redirecting the ESD. A standard practice is to place a ground ring on the perimeter of the circuit board, as [Figure 347](#) shows. The ground ring should connect to the chassis ground. Using a hatched ground plane around the button or slider sensor can also redirect the ESD event away from the sensor and PSoC™.

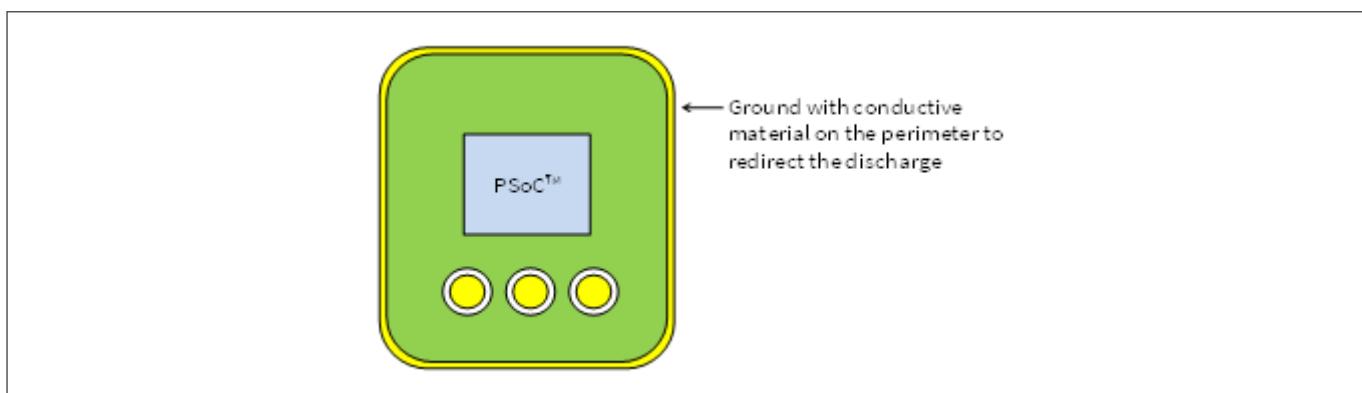


Figure 347 **Ground ring**

ESD protection devices

You can use ESD protection devices on vulnerable traces. Select ESD protection devices with a low input capacitance to avoid reduction in CAPSENSE™ sensitivity. [Table 75](#) lists the recommended ESD protection devices.

~~5 PSoC™ 6 application notes~~

~~DRAFT~~ Table 75 ESD protection devices

ESD protection device		Input capacitance	Leakage current	Contact maximum ESD limit	Air discharge maximum ESD limit
Manufacturer	Part number				
Littelfuse	SP723	5 pF	2 nA	8 kV	15 kV
Vishay	VBUS05L1-DD1	0.3 pF	0.1 µA	±15 kV	±16 kV
NXP	NUP1301	0.75 pF	30 nA	8 kV	15 kV

Electromagnetic compatibility (EMC) considerations

EMC is related to the generation, transmission, and reception of electromagnetic energy that can affect the working of an electronic system. Electronic devices are required to comply with specific limits for emitted energy and susceptibility to external events. Several regulatory bodies worldwide set regional regulations to help ensure that electronic devices do not interfere with each other.

CMOS analog and digital circuits have very high input impedance. As a result, they are sensitive to external electric fields. Therefore, you should take adequate precautions to ensure their proper operation in the presence of radiated and conducted noise.

Computing devices are regulated in the US by the FCC under Part 15, Sub-Part B for unintentional radiators. The standards for Europe and the rest of the world are adapted from CENELEC. These are covered under CISPR standards (dual-labeled as ENxxxx standards) for emissions, and under IEC standards (also dual labeled as ENxxxx standards) for immunity and safety concerns.

The general emission specification is EN55022 for computing devices. This standard cover both radiated and conducted emissions. Medical devices in the US are not regulated by the FCC, but rather are regulated by FDA rules, which include requirements of EN55011, the European norm for medical devices. Devices that include motor controls are covered under EN55014 and lighting devices are covered under EN50015.

These specifications have essentially similar performance limitations for radiated and conducted emissions. Radiated and conducted immunity (susceptibility) performance requirements are specified by several sections of EN61000-4. Line voltage transients, ESD and some safety issues are also covered in this standard.

Radiated interference and emissions

While PSoC™ 4 and PRoC Bluetooth® LE offer a robust CAPSENSE™ performance, radiated electrical energy can influence system measurements and potentially influence the operation of the CAPSENSE™ processor core. Interference enters the CAPSENSE™ device at the PCB level through sensor traces and through other digital and analog inputs. CAPSENSE™ devices can also contribute to electromagnetic compatibility (EMC) issues in the form of radiated emissions.

Use the following techniques to minimize the radiated interference and emissions.

Hardware considerations

Ground Pane

In general, proper ground plane on the PCB reduces both RF emissions and interference. However, solid grounds near CAPSENSE™ sensors or traces connecting these sensors to PSoC™ pins increase the parasitic capacitance of the sensors. It is thus recommended to use hatched ground planes surrounding the sensor and on the bottom layer of the PCBs, below the sensors, as explained in the [Ground plane](#) section in [PCB layout guidelines](#). Solid ground may be used below the device and other circuitry on the PCB which is farther from

5 PSoC™ 6 application notes

CAPSENSE™ sensors and traces. A solid ground flood is not recommended within 1 cm of CAPSENSE™ sensors or traces.

Series resistors on CAPSENSE pins

Every CAPSENSE™ controller pin has some parasitic capacitance (C_p) associated with it. As Figure 348 shows, adding an external resistor forms a low-pass RC filter that attenuates the RF noise amplitude coupled to the pin. This resistance also forms a low-pass filter with the parasitic capacitance of the CAPSENSE™ sensor that significantly reduces the RF emissions.

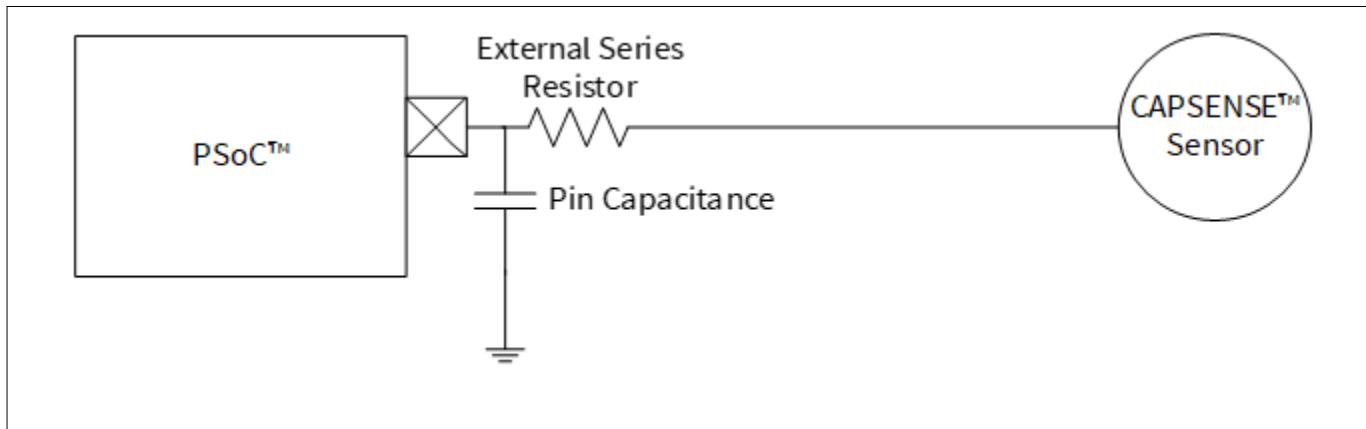


Figure 348 **RC filter**

Series resistors should be placed close to the device pins so that the radiated noise picked by the traces gets filtered at the input of the device. Thus, it is recommended to place series resistors within 10 mm of the pins.

For CAPSENSE™ designs using copper on PCBs, the recommended series resistance for CAPSENSE™ input lines is $560\ \Omega$. Adding resistance increases the time constant of the switched-capacitor circuit that converts C_p into an equivalent resistor; see [GPIO cell capacitance to current converter](#). If the series resistance value is larger than $560\ \Omega$, the slower time constant of the switching circuit suppresses the emissions and interference, but limits the amount of charge that can transfer. This lowers the signal level, which in turn lowers the [SNR](#). Smaller values are better in terms of SNR, but are less effective at blocking RF.

Series resistors on digital communication lines

Communication lines, such as I²C and SPI, also benefit from series resistance; $330\ \Omega$ is the recommended value for series resistance on communication lines. Communication lines have long traces that act as antennae similar to the CAPSENSE™ traces. The recommended pull-up resistor value for I²C communication lines is $4.7\ k\Omega$. If more than $330\ \Omega$ is placed in series on these lines, the V_{IL} and V_{IH} voltage levels may fall out of specifications. $330\ \Omega$ will not affect I²C operation as the V_{IL} level still remains within the I²C specification limit of $0.3\ V_{DD}$ when PSoC™ outputs a LOW.

5 PSoC™ 6 application notes

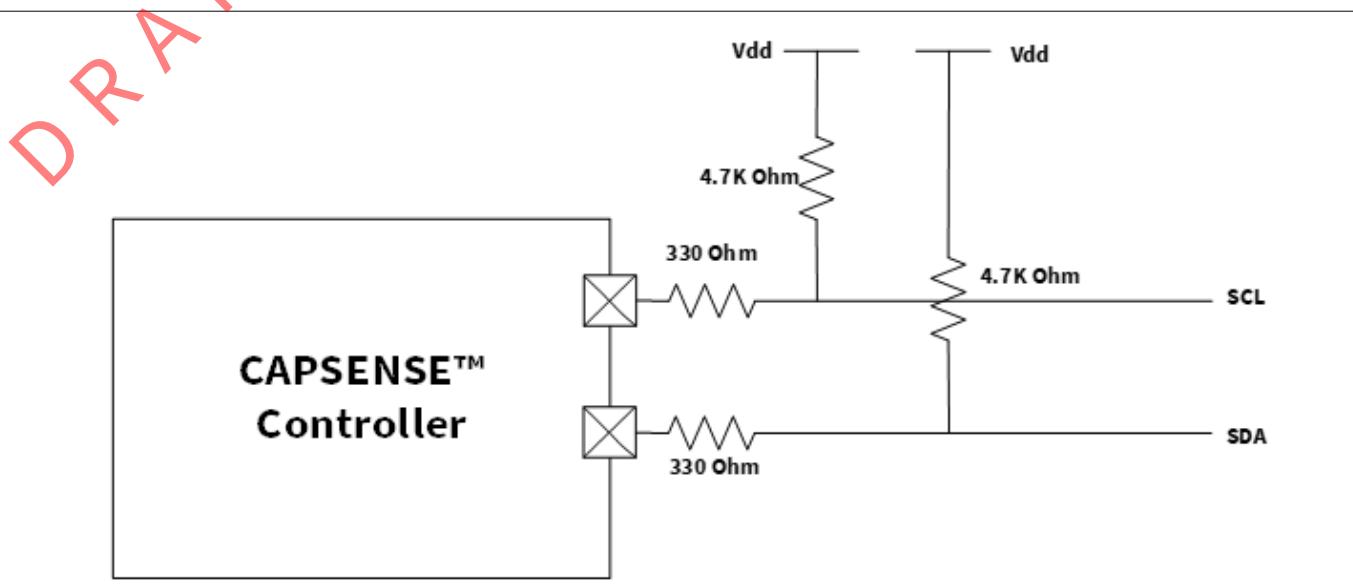


Figure 349 Series resistors on communication lines

Trace length

Long traces can pick up more noise than short traces. Long traces also add to C_P . Minimize the trace length whenever possible.

Current loop area

Another important layout consideration is to minimize the return path for currents. This is important as the current flows in loops. Unless there is a proper return path for high-speed signals, the return current will flow through a longer return path forming a larger loop, thus leading to increased emissions and interference.

If you isolate the CAPSENSE™ ground hatch and the ground fill around the device, the sensor-switching current may take a longer return path, as [Figure 350](#) shows. As the CAPSENSE™ sensors are switched at a high frequency, the return current may cause serious EMC issues. Therefore, you should use a single ground hatch, as [Figure 351](#) shows.

5 PSoC™ 6 application notes

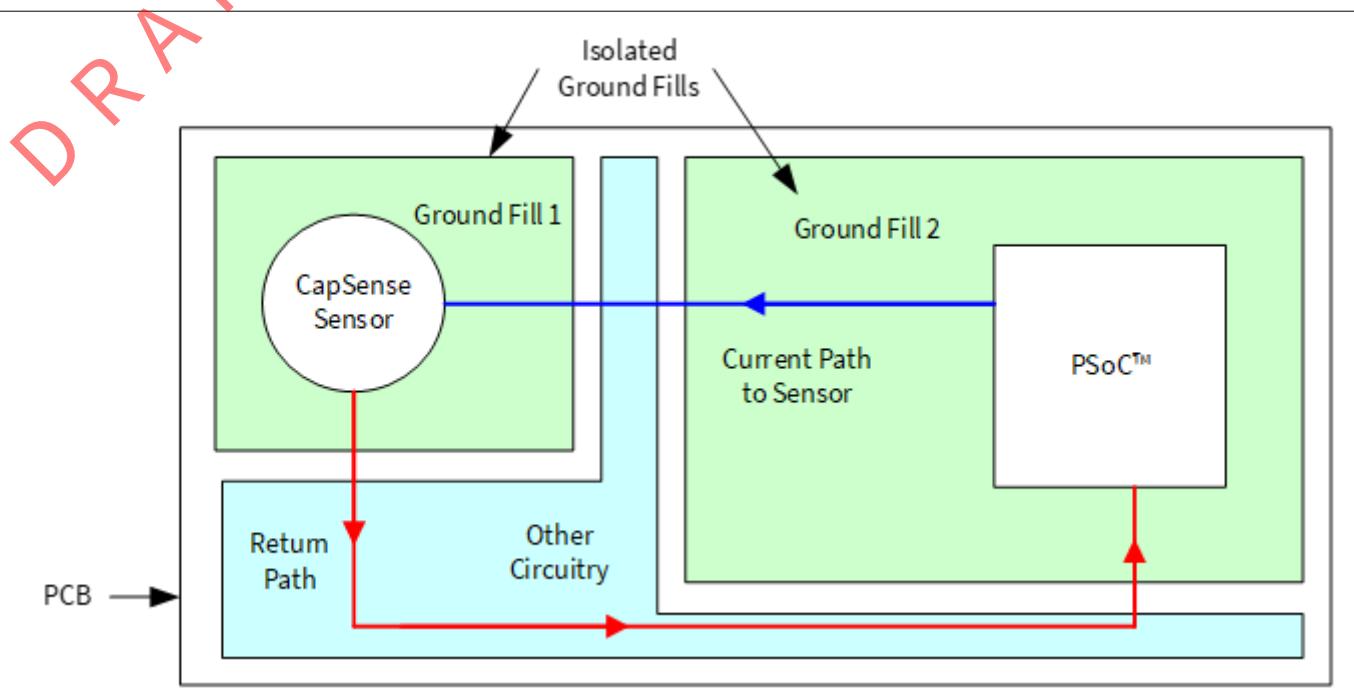


Figure 350 Improper current loop layout

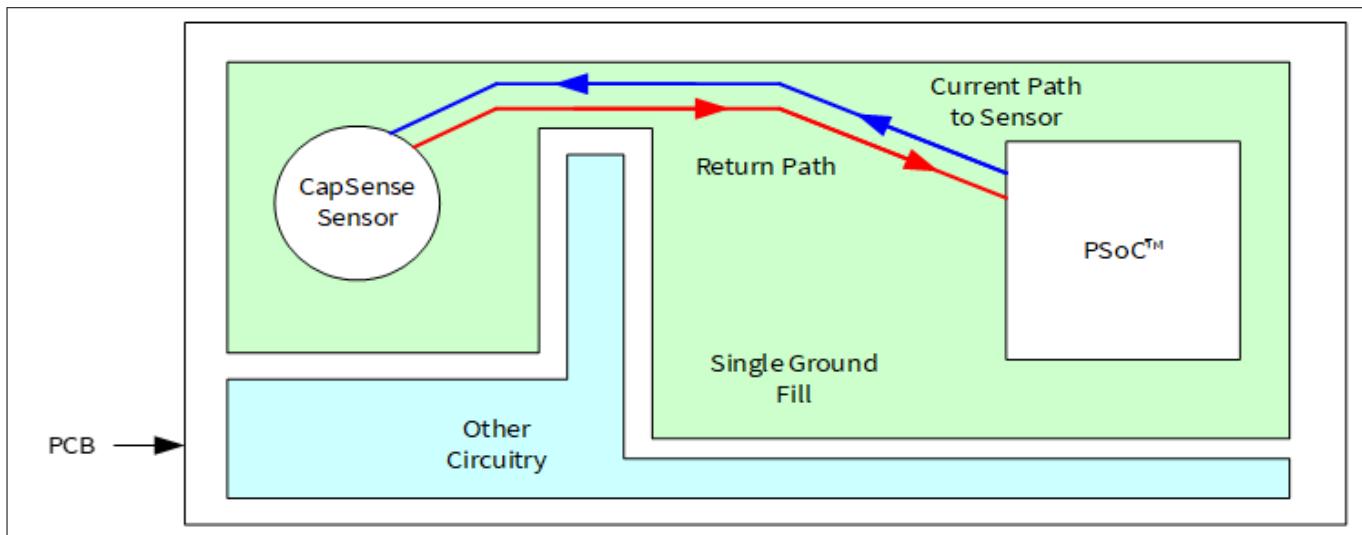


Figure 351 Proper current loop layout

RF source location

If your system has a circuit that generates RF noise, such as a switched-mode power supply (SMPS) or an inverter, you should place these circuits away from the CAPSENSE™ interface. You should also shield such circuits to reduce the emitted RF. [Figure 352](#) shows an example of separating the RF noise source from the CAPSENSE™ interface.

5 PSoC™ 6 application notes

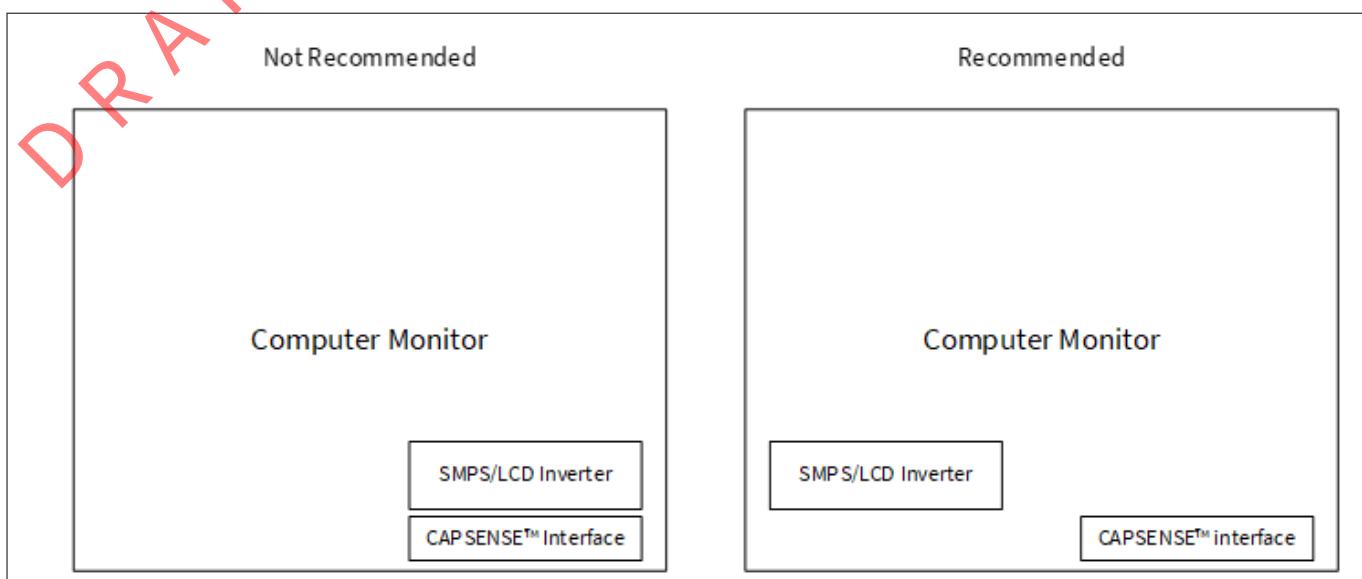


Figure 352 Separating noise sources

Firmware considerations

The following parameters affect Radiated Emissions (RE) in a CAPSENSE™ system:

- Device operating voltage
- Device operation frequency
- Sensor switching frequency
- Shield signal
- Sensor scan time
- Sense Clock Source Inactive sensor termination

The following sections explain the effect of each parameter.

Device operating voltage

The emission is directly proportional to the voltage levels at which switching happens. Reducing the operating voltage helps to reduce the emissions as the amplitude of the switching signal at any output pin directly depends on the operating voltage of the device.

PSoC™ allows you to operate at lower operating voltages, thereby reducing the emissions. [Figure 353](#) and [Figure 354](#) show the impact of operating voltage on radiated emissions. Because IMO = 24 MHz, there is a spike at 24 MHz and the other spikes are caused by different hardware and firmware operations of the device.

5 PSoC™ 6 application notes

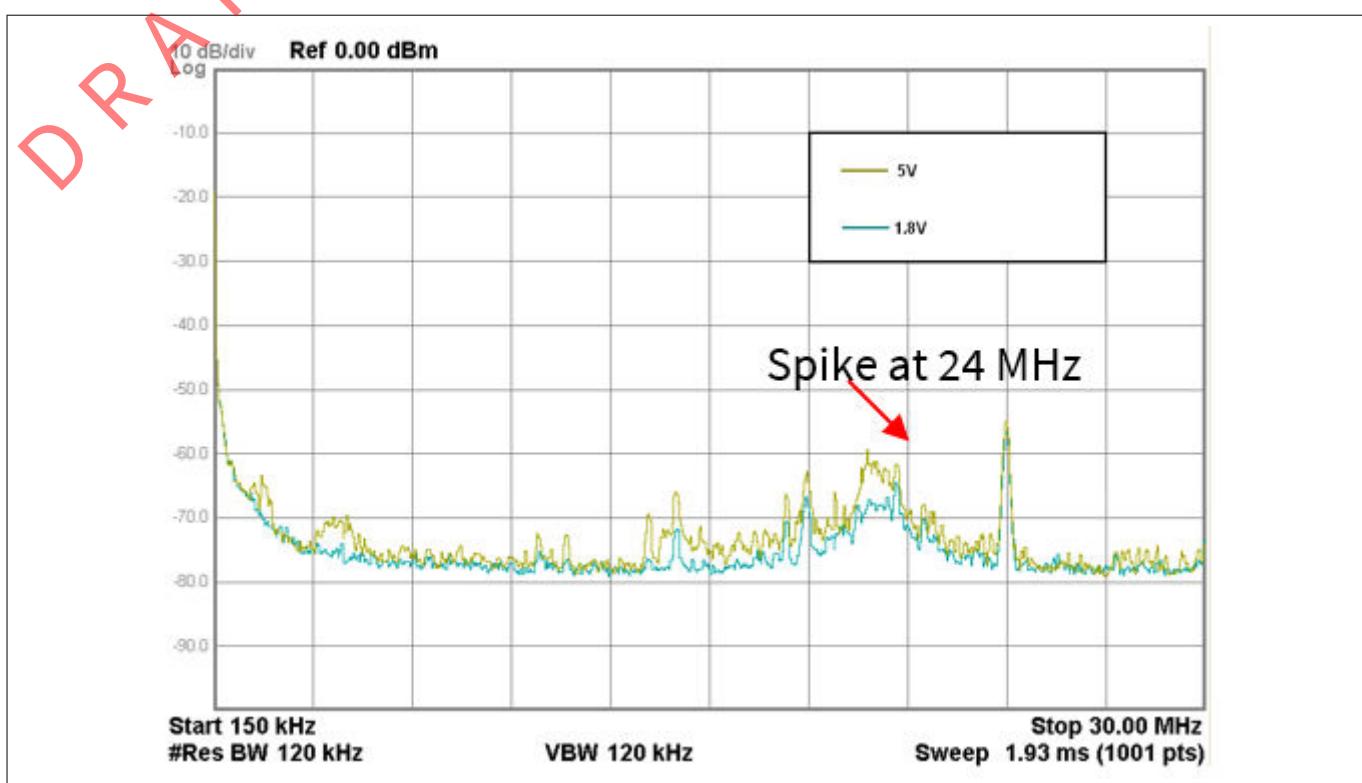


Figure 353 Effect of V_{DD} on radiated emissions (150 kHz – 30 MHz)

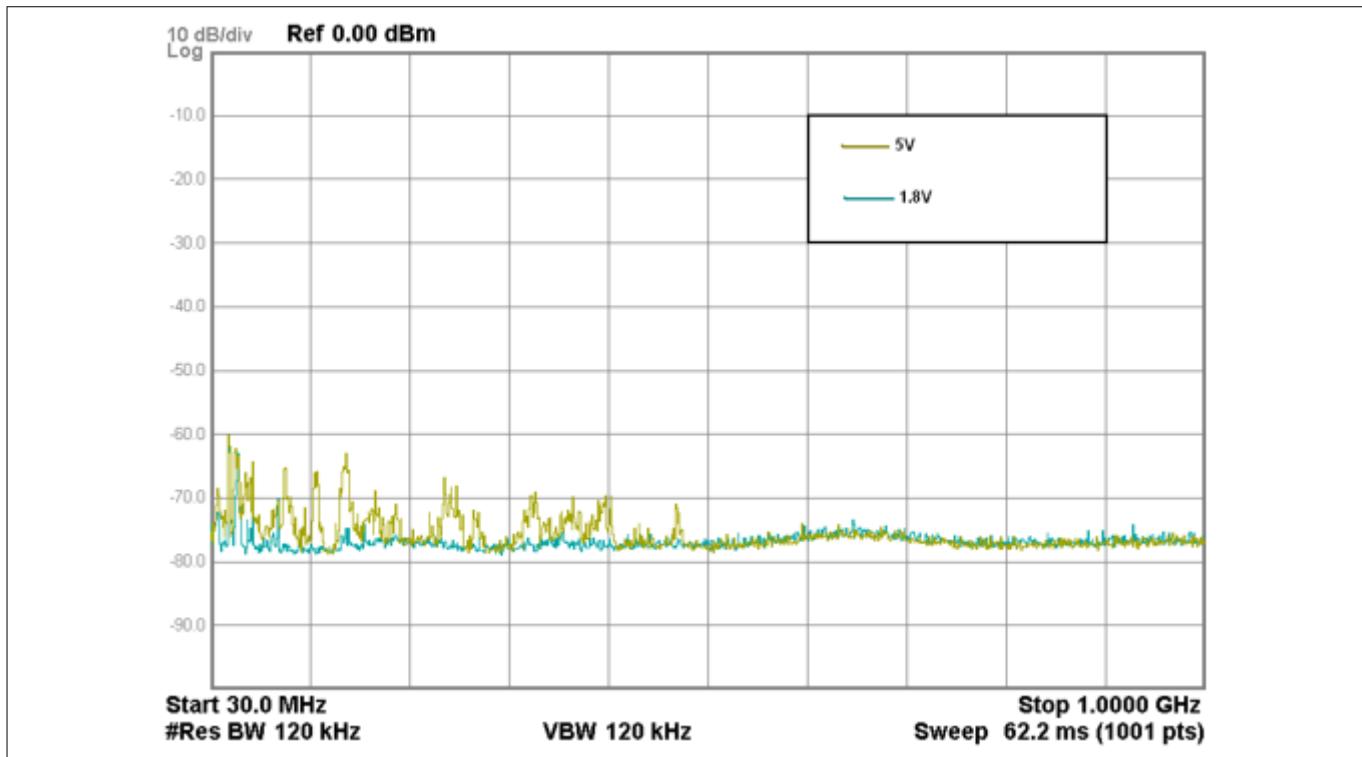


Figure 354 Effect of V_{DD} on radiated emissions (30 MHz – 1 GHz)

Note: Frequency axis is in log scale.

5 PSoC™ 6 application notes

Device operating frequency

Reducing the system clock frequency (IMO frequency) reduces radiated emissions. However, reducing the IMO frequency may not be feasible in all applications because the IMO frequency impacts the CPU clock and all other system timings. Choose a suitable IMO frequency based on your application.

Sensor-switching frequency

Reducing the sensor-switching frequency (see [Sense Clock](#)) also helps to reduce radiated emissions. See [Figure 355](#) and [Figure 356](#). Because IMO = 24 MHz, there is a spike at 24 MHz and the other spikes are caused by different hardware and firmware operations of the device.

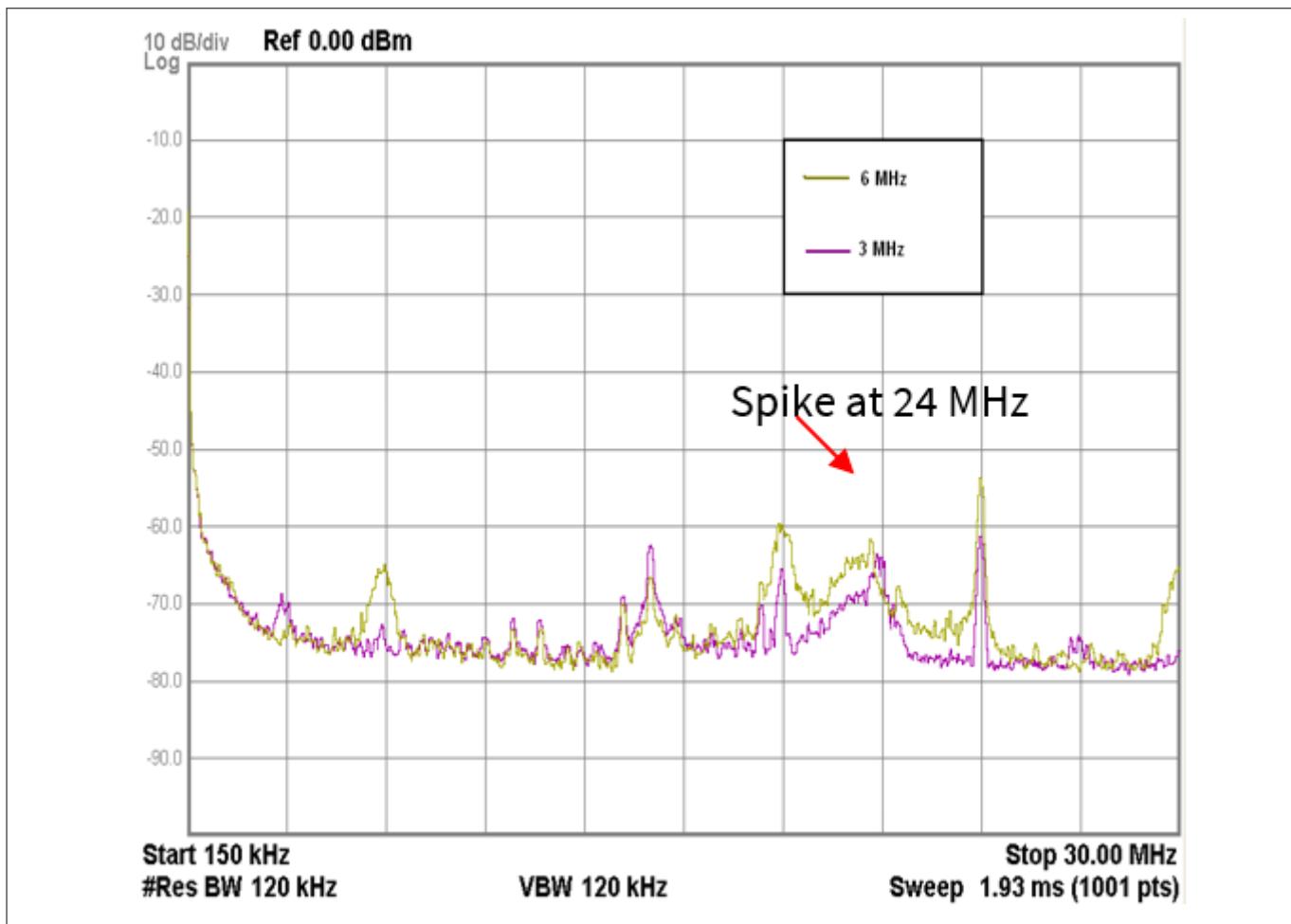


Figure 355 Effect of sensor-switching frequency on radiated emissions (150 kHz – 30 MHz)

5 PSoC™ 6 application notes

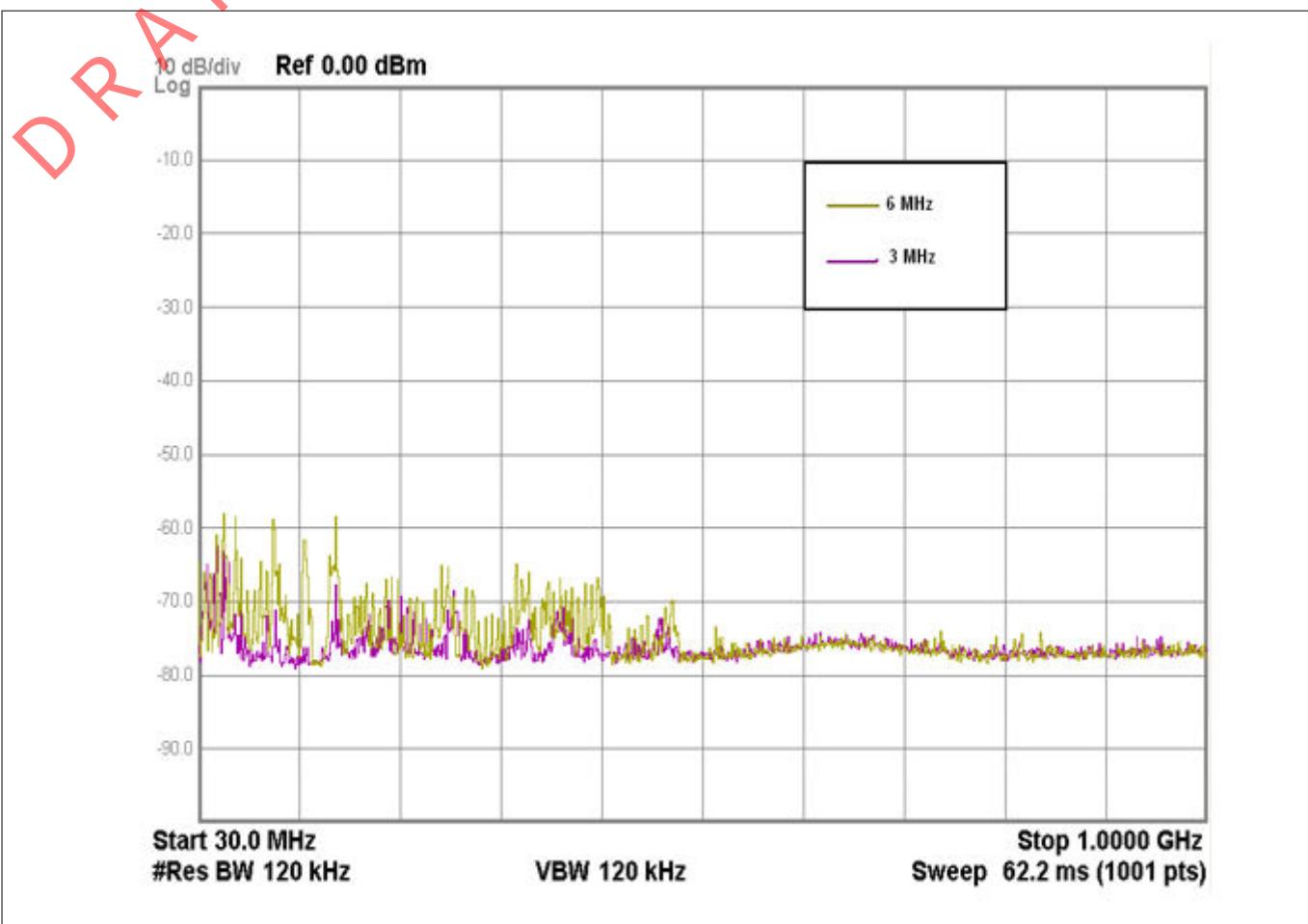


Figure 356 Effect of sensor-switching frequency on radiated emissions (30 MHz – 1 GHz)

Note: Frequency axis is in log scale.

Pseudo random sense clock

The PSoC™ 4 device supports PRS-based sense clock generation. A PRS is used instead of a fixed clock source to attenuate emitted noise on the CAPSENSE™ pins by reducing the amount of EMI created by a fixed-frequency source and to increase EMI immunity from other sources and their harmonics.

Spread spectrum sense clock

In addition to the PRS-based clock generation, the PSoC™ 4 S-Series, PSoC™ 4100S Plus, PSoC™ 4100PS, and PSoC™ 6 MCU family of devices supports a unique feature called spread spectrum sense clock generation, in which the sense clock frequency is spread over a desired range. This method will help to reduce the peaks and spread out the emissions over a range of frequencies. The spread spectrum clock can be enabled by selecting the **Sense Clock Source** as **SSCn**. The range of frequency spread is decided by the length of the register. For more details on the spread spectrum clock generation in the PSoC™ 4 S-Series, PSoC™ 4100S Plus, and PSoC™ 4100PS family, see the Spread spectrum clock section in the CAPSENSE™ chapter of the respective device [Technical reference manual](#).

5 PSoC™ 6 application notes

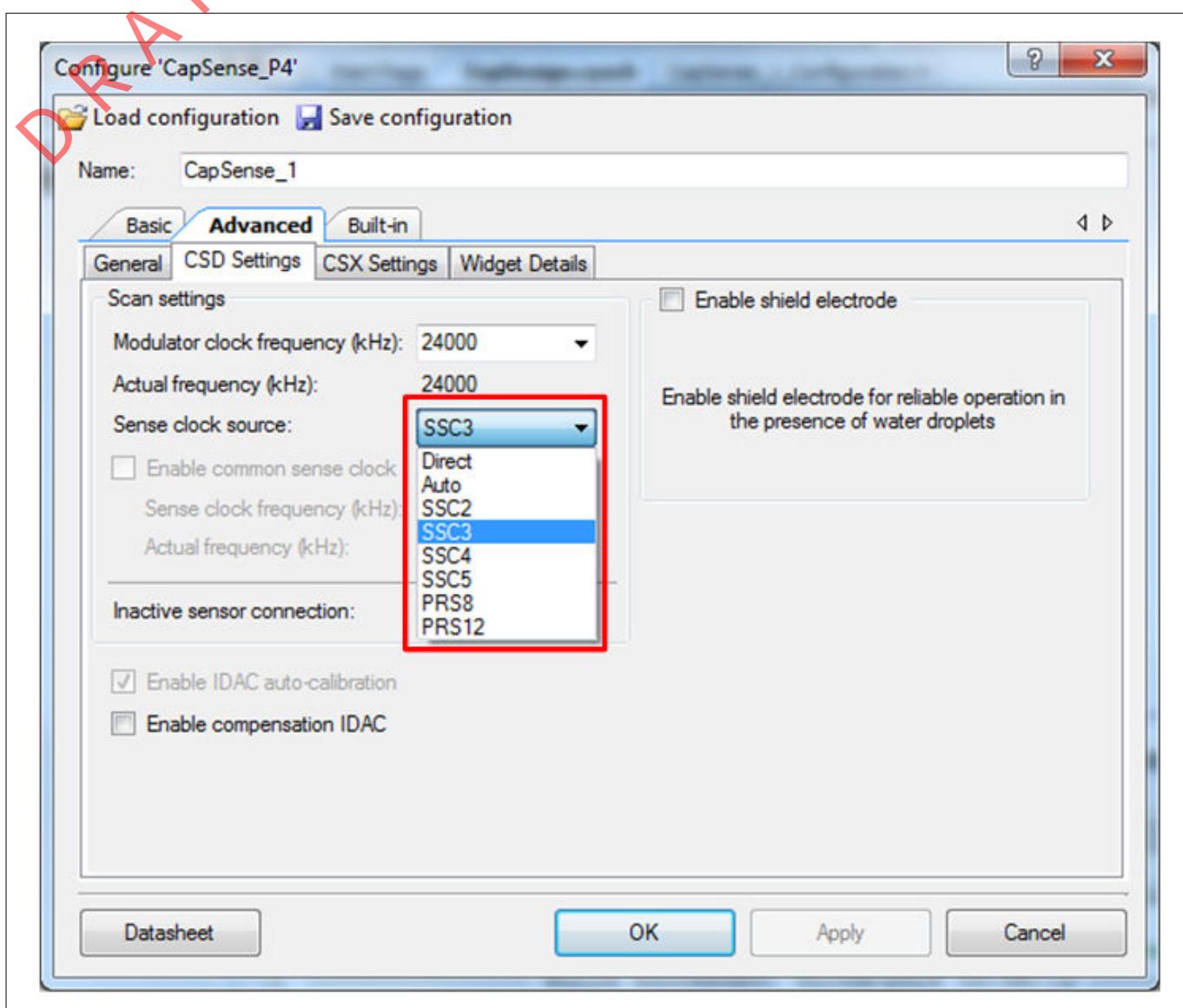


Figure 357 Sense clock sources in PSoC™ 4 S-Series, PSoC™ 4100S Plus, and PSoC™ 4100PS family

Shield signal

Shield signal

Enabling the shield signal (see [Driven shield signal and shield electrode](#)) on the hatch pattern increases the radiated emissions. Enable the driven-shield signal only for liquid-tolerant, proximity-sensing, or high-parasitic-capacitance designs. Also, if the shield must be used, ensure that the shield electrode area is limited to a width of 1 cm from the sensors, as [Figure 338](#) shows.

[Figure 358](#) and [Figure 359](#) show the impact of enabling the driven-shield signal on the hatch pattern surrounding the sensors on radiated emissions.

Note: *In these figures, the hatch pattern is grounded when the driven-shield signal is disabled. Because $f_{IMO} = 24\text{ MHz}$, there is a spike at 24 MHz and the other spikes are caused by different hardware and firmware operations of the device.*

5 PSoC™ 6 application notes

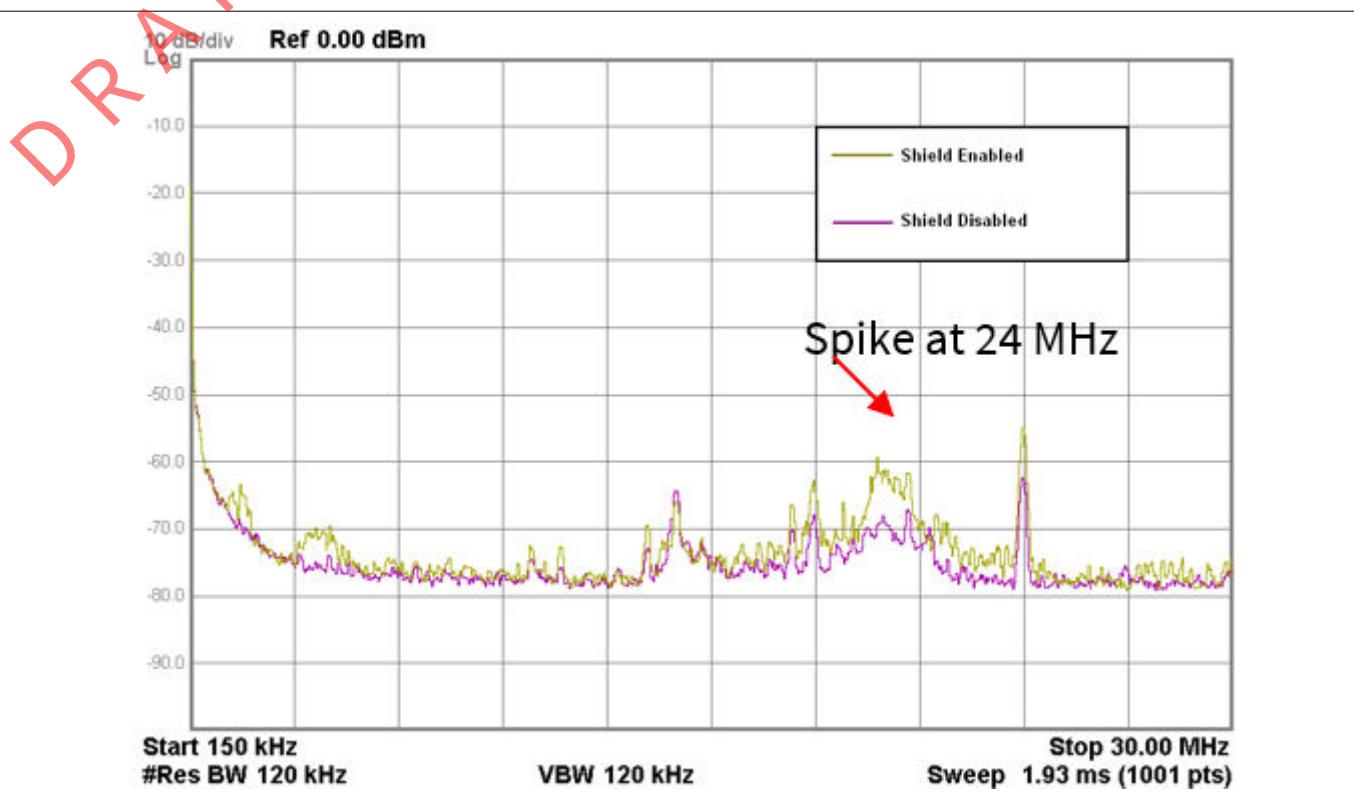


Figure 358 Effect of shield electrode on radiated emissions (150 kHz – 30 MHz)

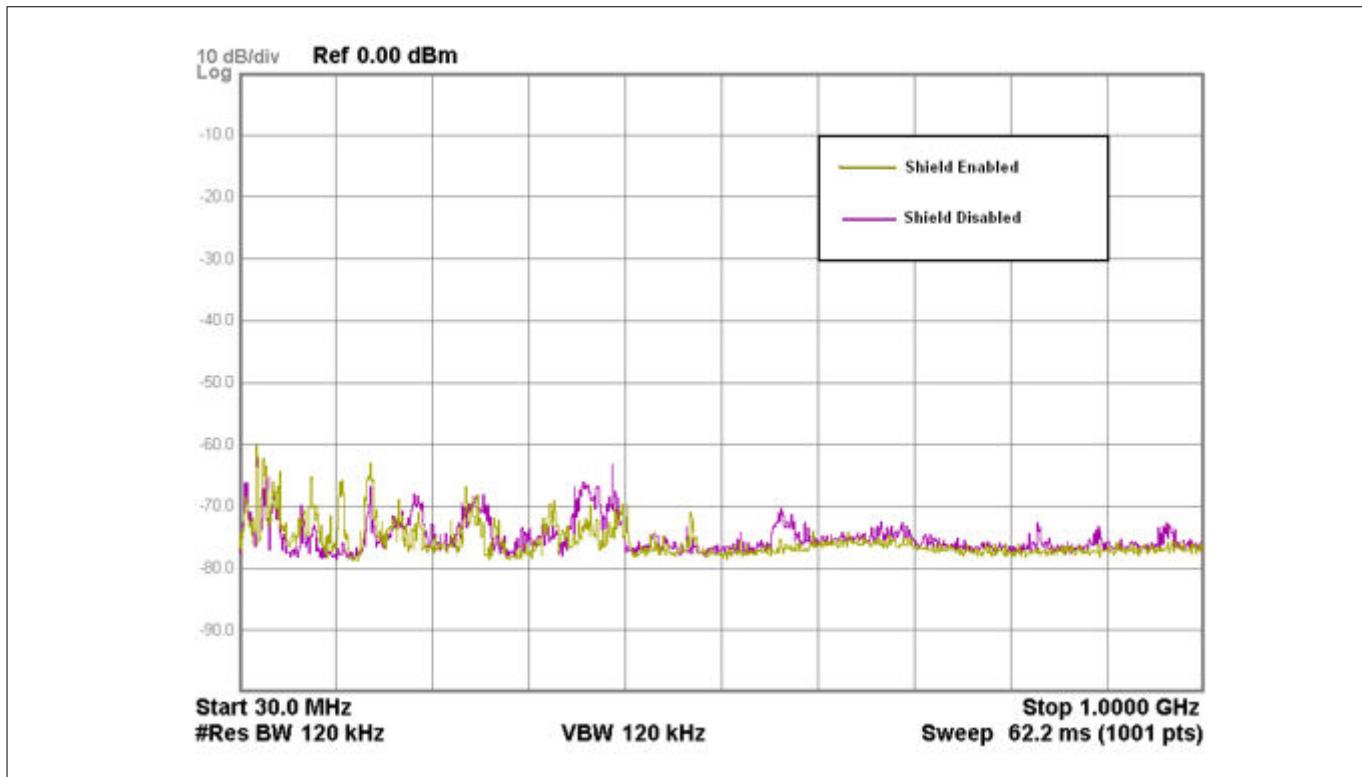


Figure 359 Effect of shield electrode on radiated emissions (30 MHz – 1 GHz)

Note: Frequency axis is in log scale.

~~5 PSoC™ 6 application notes~~

~~Sensor scan time~~

Reducing the sensor scan time reduces the average radiated emissions. The sensor-scan time depends on the scan resolution and modulator clock divider (see [Equation 10](#)). Increasing the scan resolution or modulator clock divider increases the scan time.

[Figure 360](#) and [Figure 361](#) show the impact of sensor scan time on radiated emissions. Note that, here, the sensor scan time was varied by changing the scan resolution. Because IMO = 24 MHz, there is a spike at 24 MHz and the other spikes are caused by different hardware and firmware operations of the device.

Table 76 Sensor scan time

Parameter	Total scan time for five buttons	
	0.426 ms	0.106 ms
Modulation clock divider	2	2
Scan resolution	10 bits	8 bits
Individual sensor scan time	0.085 ms	0.021 ms

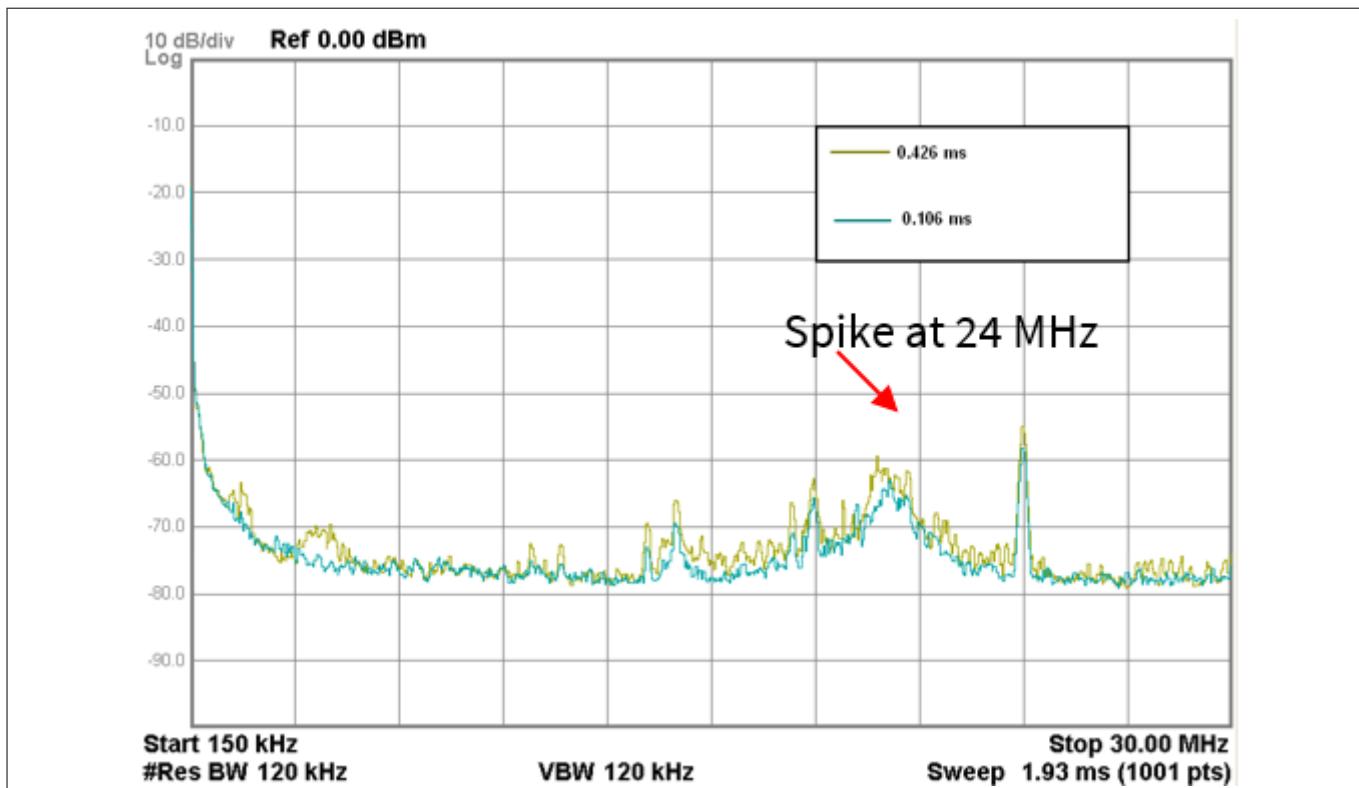
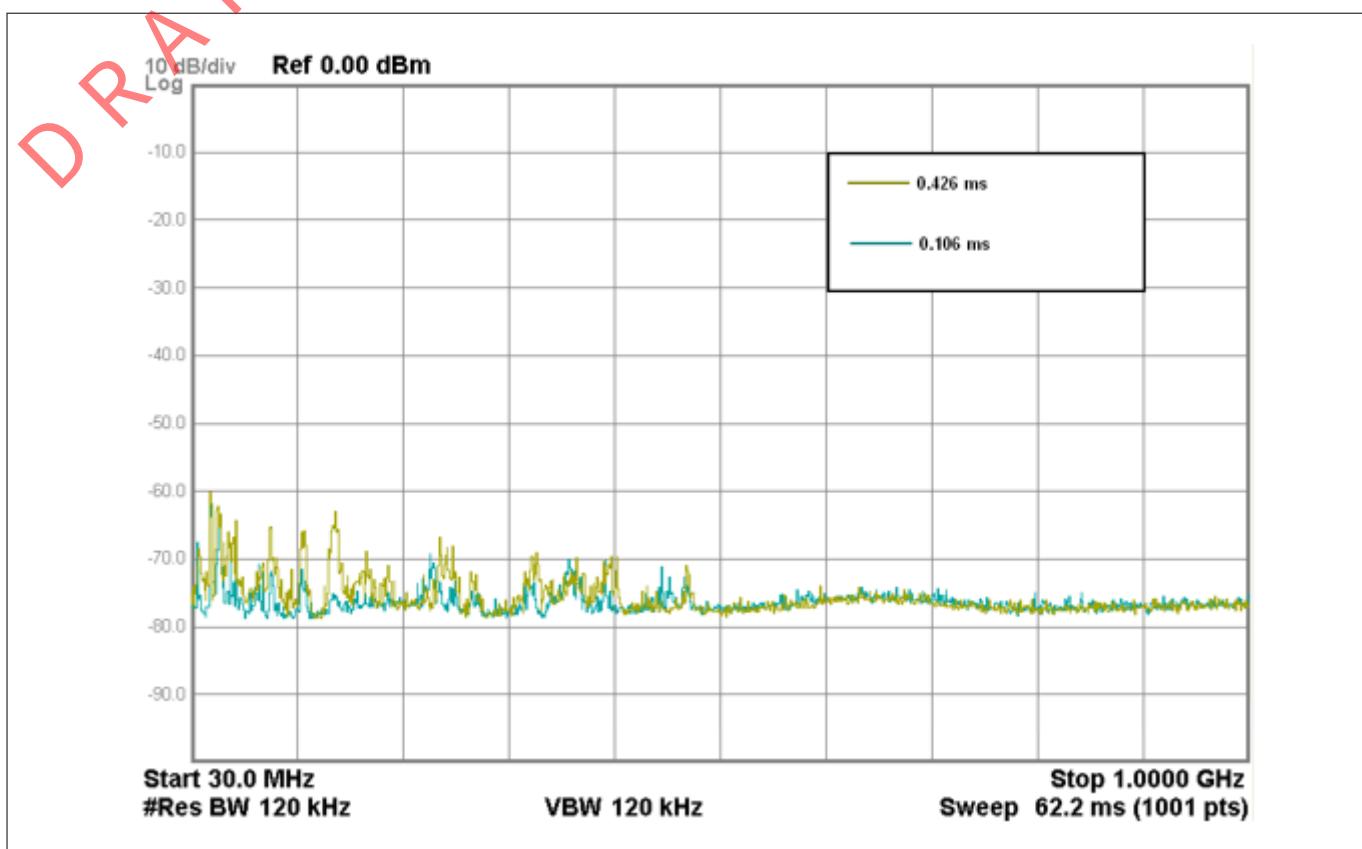


Figure 360 Effect of scan time on radiated emissions (150 kHz – 30 MHz)

5 PSoC™ 6 application notes**Figure 361 Effect of scan time on radiated emissions (30 MHz – 1 GHz)**

Note: Frequency axis is in log scale.

Sense clock source

Using PRS instead of direct clock drive as sense clock source spreads the radiated spectrum and hence reduces the average radiated emissions. See [Figure 362](#) and [Figure 363](#). Because IMO = 24 MHz, there is a spike at 24 MHz and the other spikes are caused by different hardware and firmware operations of the device.

5 PSoC™ 6 application notes

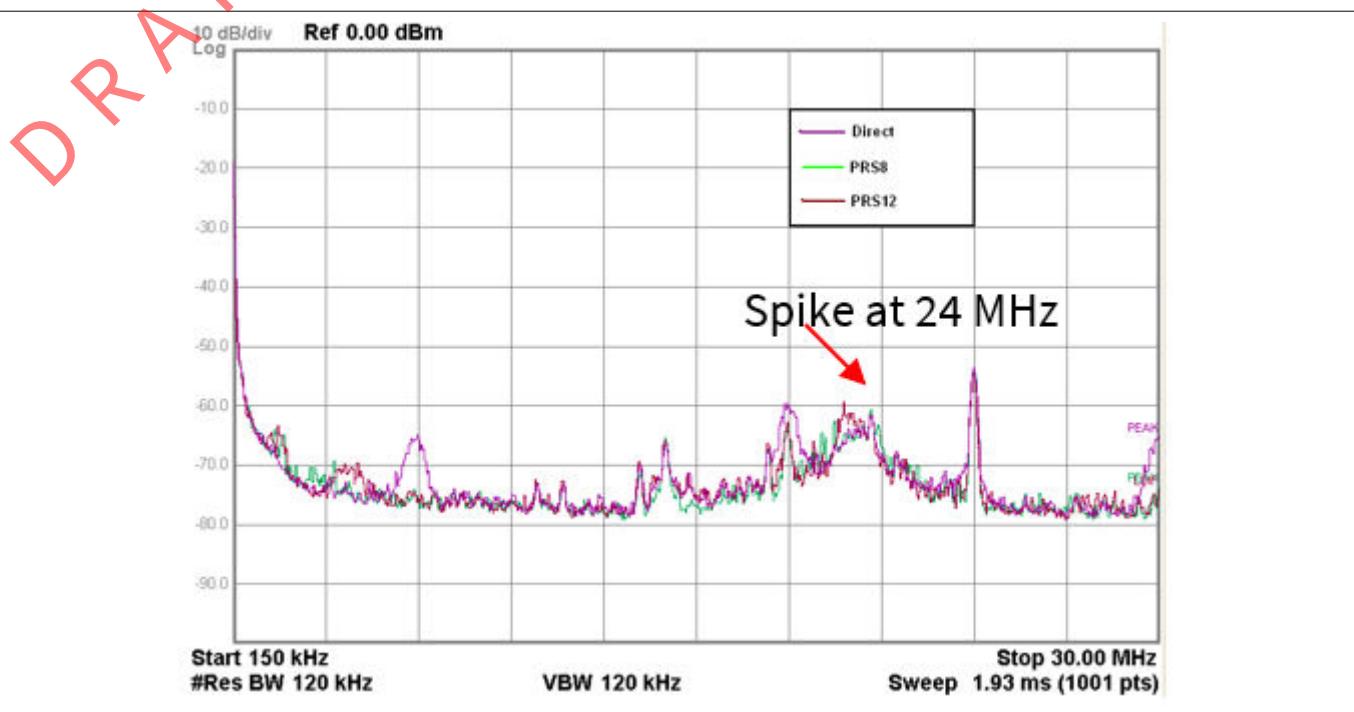


Figure 362 Effect of sense clock source on radiated emissions (150 kHz – 30 MHz)

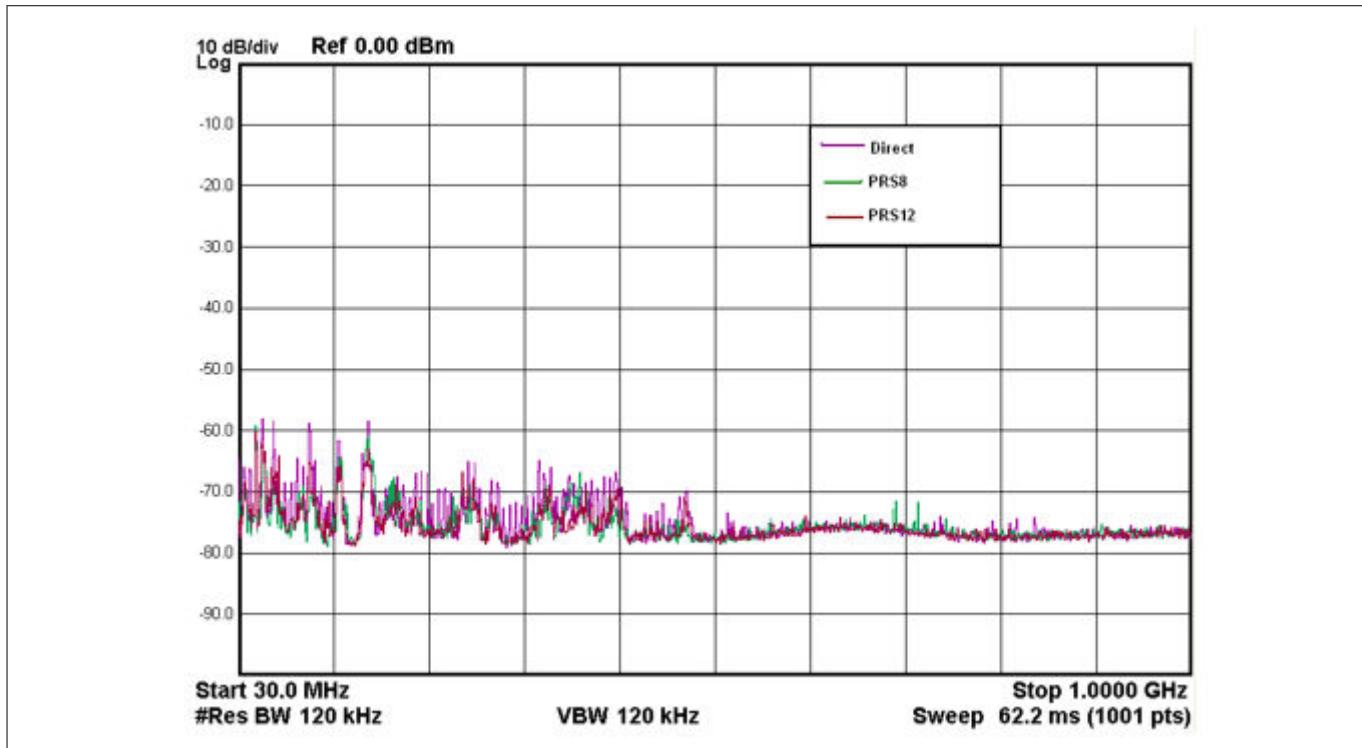


Figure 363 Effect of sense clock source on radiated emissions (30 MHz – 1 GHz)

Note: Frequency axis is in log scale.

~~5 PSoC™ 6 application notes~~~~Inactive sensor termination~~

Connecting inactive sensors to ground reduces the radiated emission by a greater degree than connecting them to the shield. [Figure 364](#) and [Figure 365](#) show the impact of different inactive sensor terminations on radiated emission. Because IMO = 24 MHz, there is a spike at 24 MHz and the other spikes are caused by different hardware and firmware operations of the device.

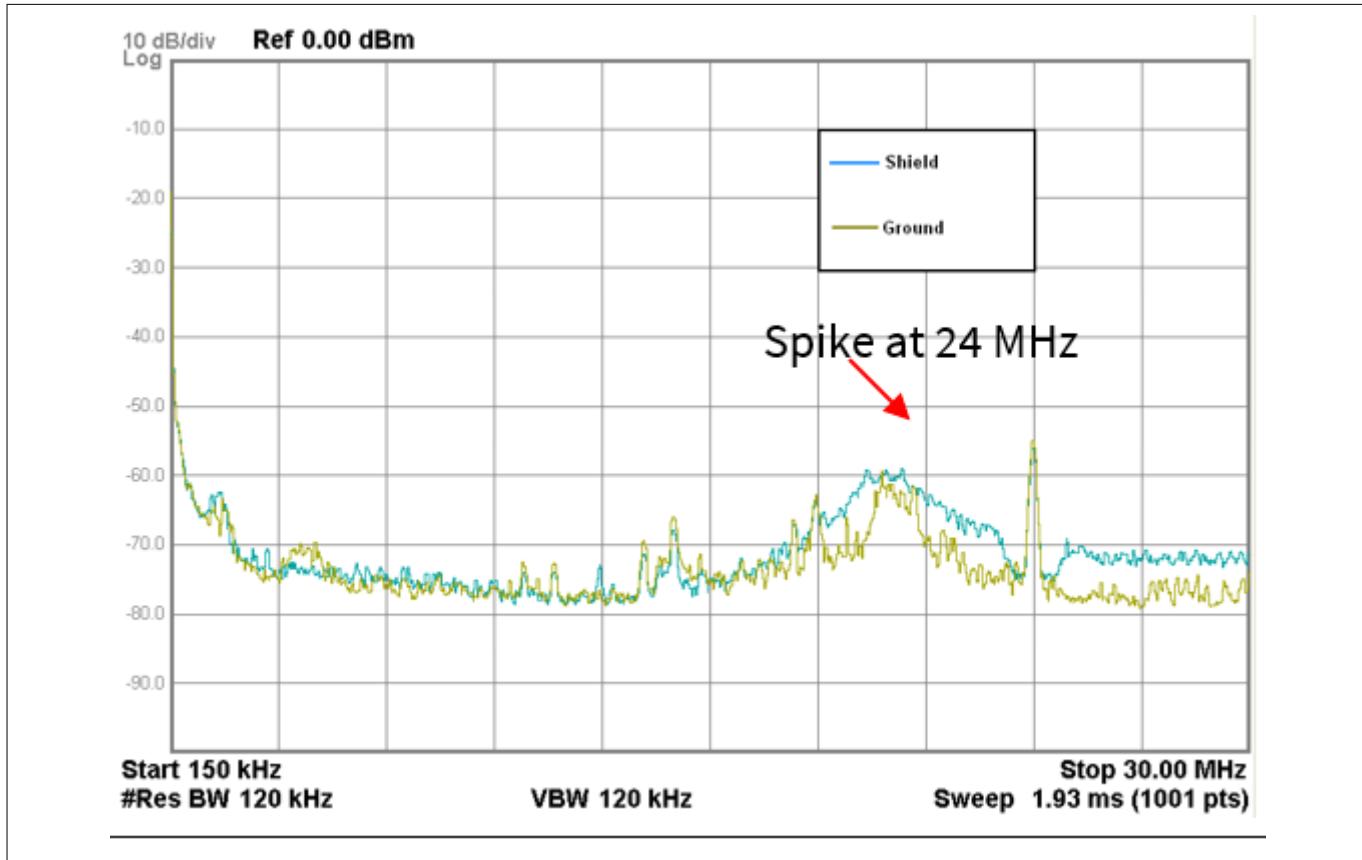


Figure 364 Effect of inactive sensor termination on radiated emissions (150 kHz – 30 MHz)

5 PSoC™ 6 application notes

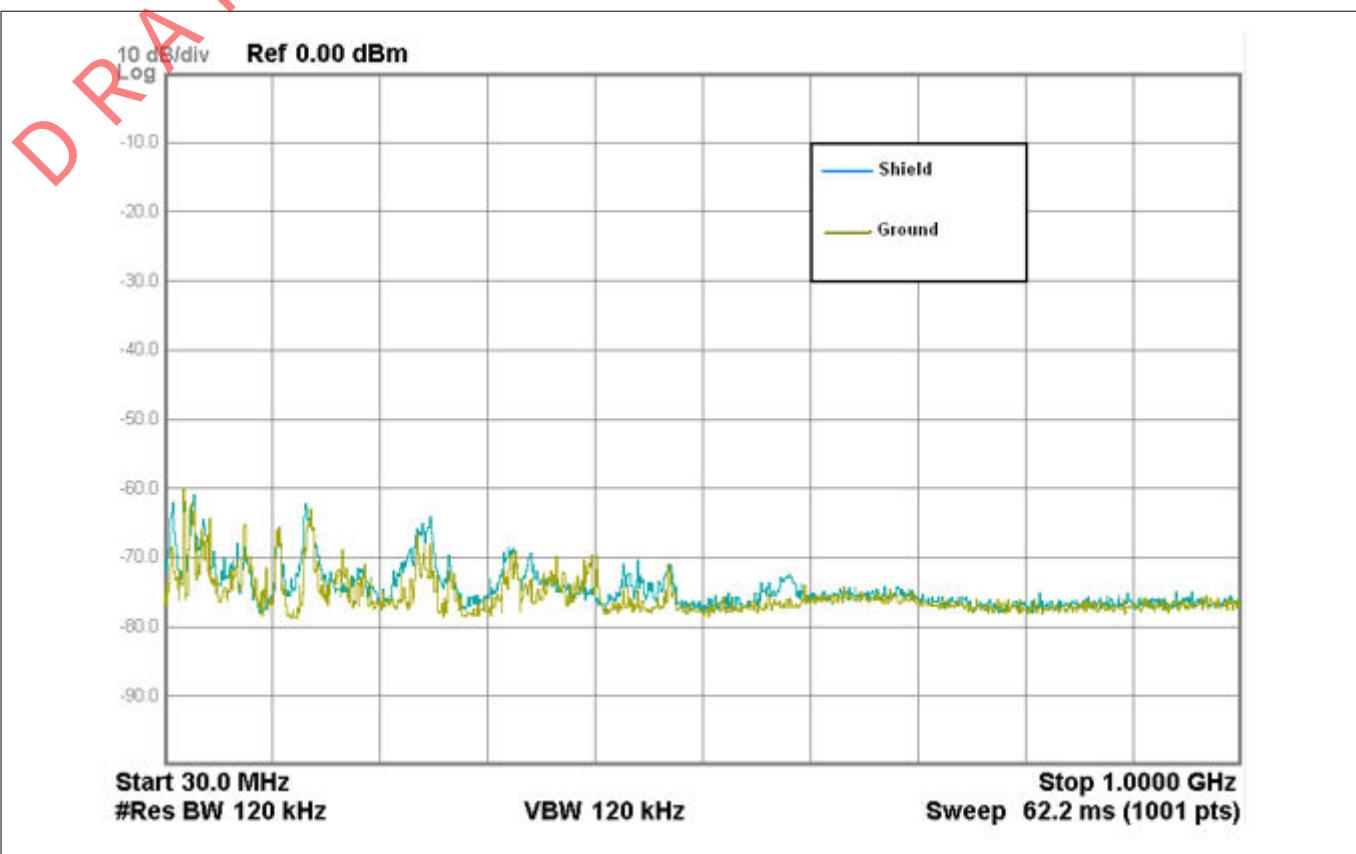


Figure 365 Effect of inactive sensor termination on radiated emissions (30 MHz – 1 GHz)

Note: Frequency axis is in log scale.

Conducted RF noise

The noise current that enters the CAPSENSE™ system through the power and communication lines is called conducted noise. You can use the following techniques to reduce the conducted RF noise.

- Use decoupling capacitors on the power supply pins to reduce the conducted noise from the power supply. See section [Power supply layout recommendations](#) and the [Device datasheet](#) for details
- Provide GND and VDD planes on the PCB to reduce current loops
- If the PSoC™ PCB is connected to the power supply using a cable, minimize the cable length and consider using a shielded cable

To reduce high-frequency noise, place a ferrite bead around power supply or communication lines.

5.8.7.6 Effect of grounding

5.8.7.6.1 CSX method

The equivalent capacitances formed in the CSX method when a finger touches the CSX sensor is shown in [Figure 366](#). From [Figure 366](#), current drawn from the IDAC (IRX) has two components: I_{mt} and I_{sc} . These two components depend on the ratio of C_{bodyDG}/C_{fs} . Because the raw count depends on the amount of current drawn from IDAC, the increase and decrease of C_{bodyDG}/C_{fs} will affect the raw count of the sensor and cause a sudden change in the behavior on some conditions. To understand it better, consider two extreme conditions which cause $C_{bodyDG} \gg C_{fs}$ and $C_{bodyDG} \ll C_{fs}$.

5 PSoC™ 6 application notes

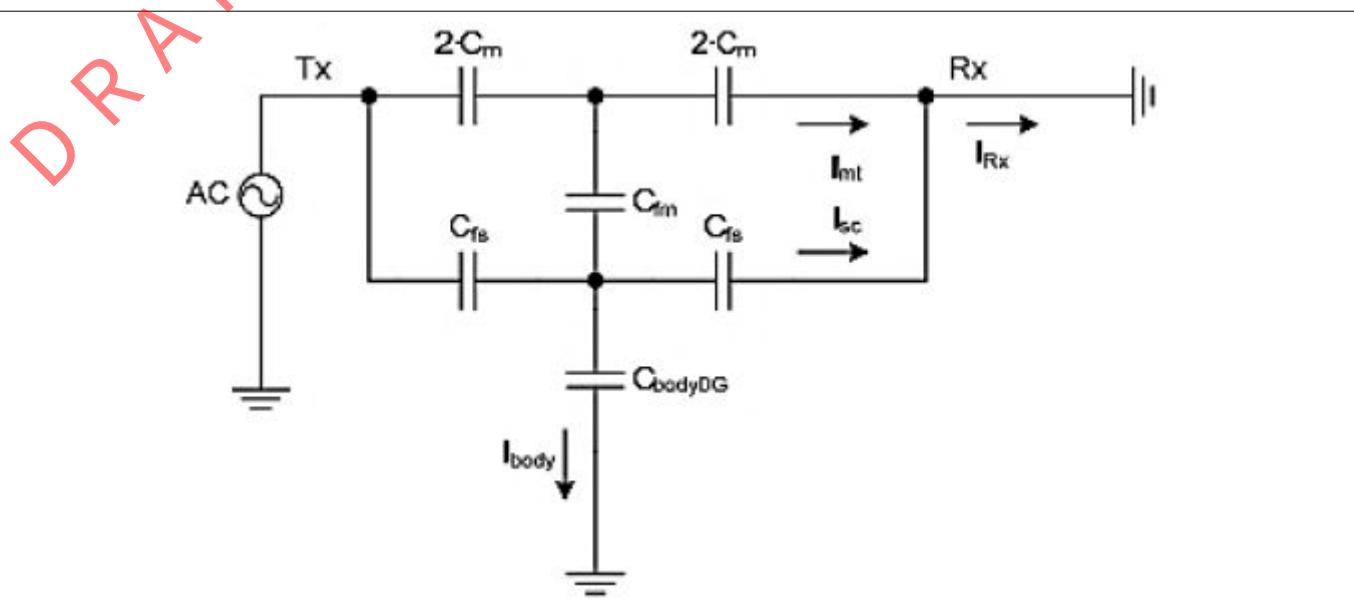


Figure 366 **Equivalent circuit of the CSX sensor when finger is placed on the button**

Where,

C_M = Mutual capacitance between the Rx and Tx electrode

C_{fs} = Capacitance formed between the surface of the finger and electrode

C_{fm} = Virtual capacitance which reduces the mutual-capacitance C_M due to placing a finger

C_{bodyDG} = Body capacitance relative to the device ground

$$I_{RX} = I_{mt} + I_{sc}$$

Equation 79 Current drawn from IDAC in CSX method

I_{mt} is due to the effective mutual-capacitance between the Tx and Rx electrode.

I_{sc} = Parasitic current that flows due to the capacitance formed between the sensor and finger

$C_{bodyDG} \gg C_{fs}$

Because $C_{bodyDG} \gg C_{fs}$, you can replace C_{bodyDG} with a ground conductor; the resulting equivalent circuit appears as shown in [Figure 367](#). Whenever there is a finger touch, the current drawn from the IDAC is directly dependent upon the effective mutual-capacitance between the Tx and Rx. This condition is observed in a good board design.

5 PSoC™ 6 application notes

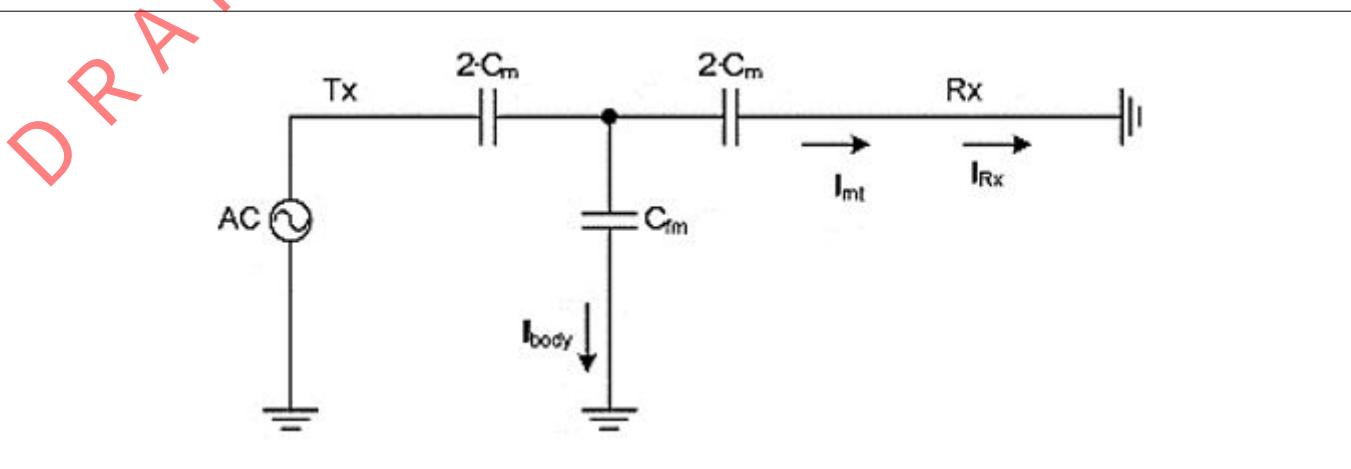


Figure 367 Equivalent circuit of the CSX sensor when $C_{\text{bodyDG}} \gg C_{\text{fs}}$

$C_{\text{bodyDG}} \ll C_{\text{fs}}$

This condition ($C_{\text{bodyDG}} \ll C_{\text{fs}}$) is observed when a finger touches a CSX button with a very thin overlay or no overlay, or a finger touching the Rx and Tx electrodes directly, or a water drop being present on the Rx and Tx electrode only. Because $C_{\text{bodyDG}} \ll C_{\text{fs}}$, you can remove C_{bodyDG} ; the equivalent circuit for this case is as shown in Figure 368. In this condition, the capacitance introduced by the finger to the electrode C_{fs} is very high compared to the capacitance of the finger relative to the device ground C_{bodyDG} .

From Figure 368, it forms a balanced bridge circuit. Due to this, no current flows through C_{fm} , and also due to increase in C_{fs} , I_{sc} increases and thus additional current is drawn from the IDAC. This causes an unexpected behavior of decrease in the raw count.

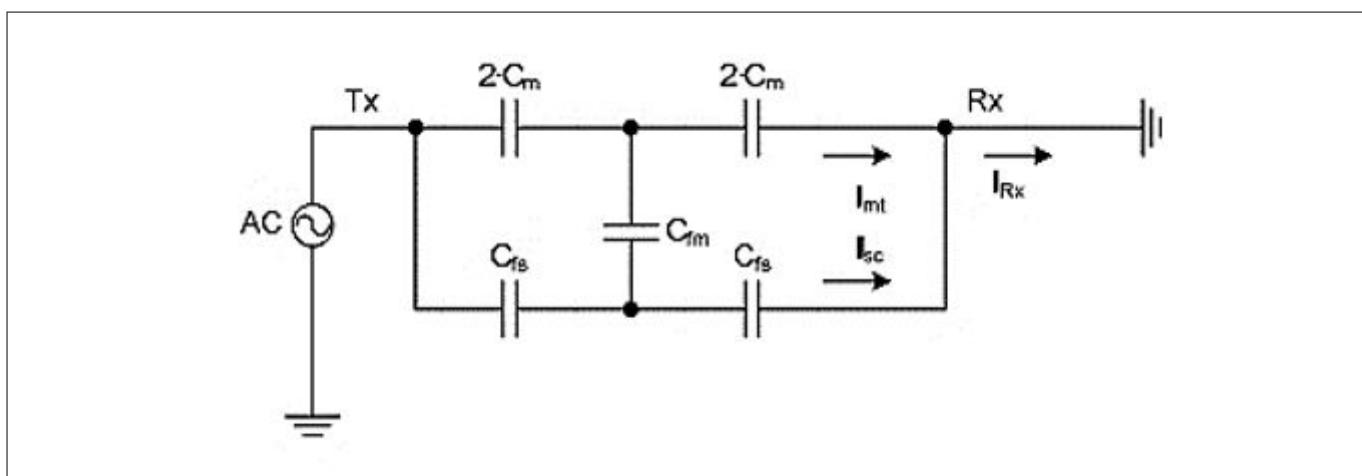


Figure 368 Equivalent circuit of the CSX sensor when $C_{\text{bodyDG}} \ll C_{\text{fs}}$

For CSX sensors, design should focus on increasing the ratio of $C_{\text{bodyDG}}/C_{\text{fs}}$. Following are the examples for increasing the ratio of $C_{\text{bodyDG}}/C_{\text{fs}}$:

1. $C_{\text{bodyDG}}/C_{\text{fs}}$ ratio depends on the thickness of the overlay, size of the sensor, and many other factors. By experimental data, you are recommended not to use overlay thickness below 0.5 mm for CSX sensor. See [Overlay thickness](#)

~~5 PSoC™ 6 application notes~~

- ~~DRAFT~~
2. If the sensor is surrounded by hatch fill connected to ground, there is a lower chance that $C_{bodyDG} \ll C_{fs}$. Therefore, ensure good ground in the design. Follow the best practices for the PCB layout guidelines described in this chapter
 3. In the design, it is recommended to isolate the trace lines of Rx and Tx electrode, external capacitors, and resistors of the CSX touch sensing system from any conducting surface or a finger touch to avoid direct interaction. Not following this recommendation may cause $C_{bodyDG} \ll C_{fs}$

5.8.7.6.2 CSD method

The equivalent capacitances formed in the CSD method when a finger touches the CSD sensor is shown in [Figure 369](#). It shows that the current drawn from the IDAC directly depends on the capacitance introduced by the finger touch. I_{CP} is a fixed component and I_{CF} depends on C_F, C_{BG}, C_{GE} . From [Sigma-delta converter](#), the raw count depends on the amount of current drawn from IDAC. To understand it better, consider two scenarios of an AC/DC mains-powered application and a battery-powered application.

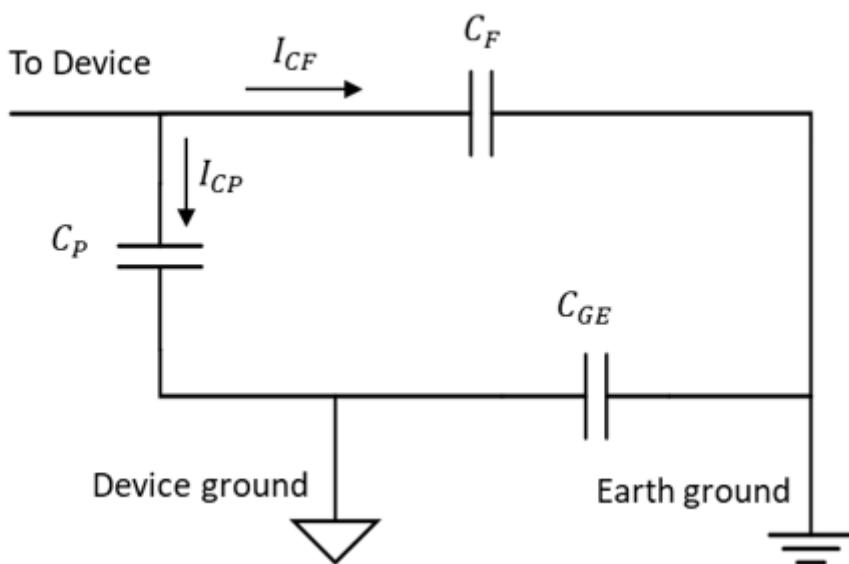


Figure 369 **Equivalent circuit of the CSD sensor**

$$I = I_{CP} + I_{CF}$$

Equation 80 **Current drawn from IDAC in CSD method**

AC/DC-powered application

In an AC / DC-powered application using the mains supply, device ground is strongly coupled to earth ground. Thus, you can replace C_{GE} with a conductor and C_{BG} is usually 100 pF to 200 pF. Since C_{BG} is large when compared to C_F , you can neglect its effect. Finally, the resulting equivalent circuit is shown in [Figure 370](#). The increase in total capacitance draws a higher current from the IDAC achieving a higher change in raw count for a finger touch. Thus, in this condition, you get a higher sensitivity, which means that you will get a higher signal for a finger touch.

5 PSoC™ 6 application notes

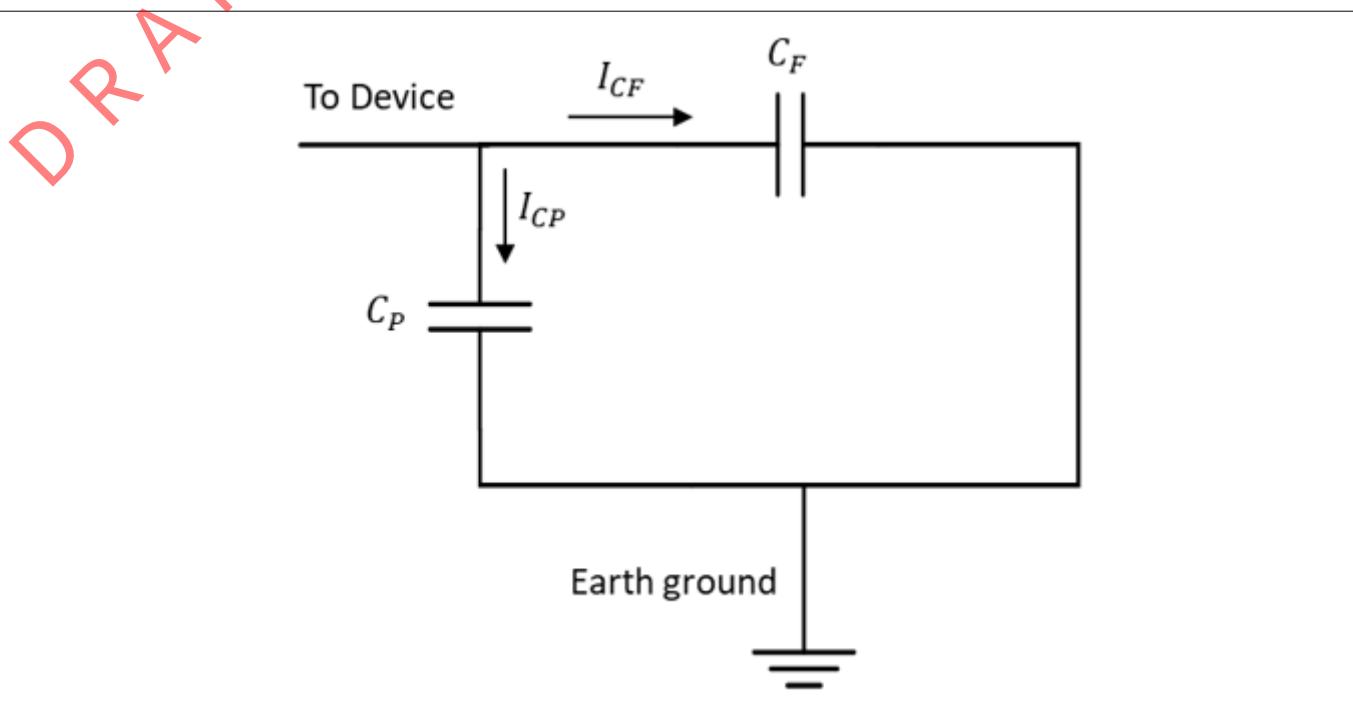


Figure 370 **Equivalent circuit of the CSD sensor for mains-powered application**

Battery-powered application

In battery-powered portable applications, device ground and earth ground are lightly coupled, thus C_{GE} is small. The resulting equivalent circuit is shown in [Figure 371](#). Thus, in this condition, you get a lower sensitivity; that means you will get a lower signal for a finger touch, which is due to a decrease in capacitance seen at the device.

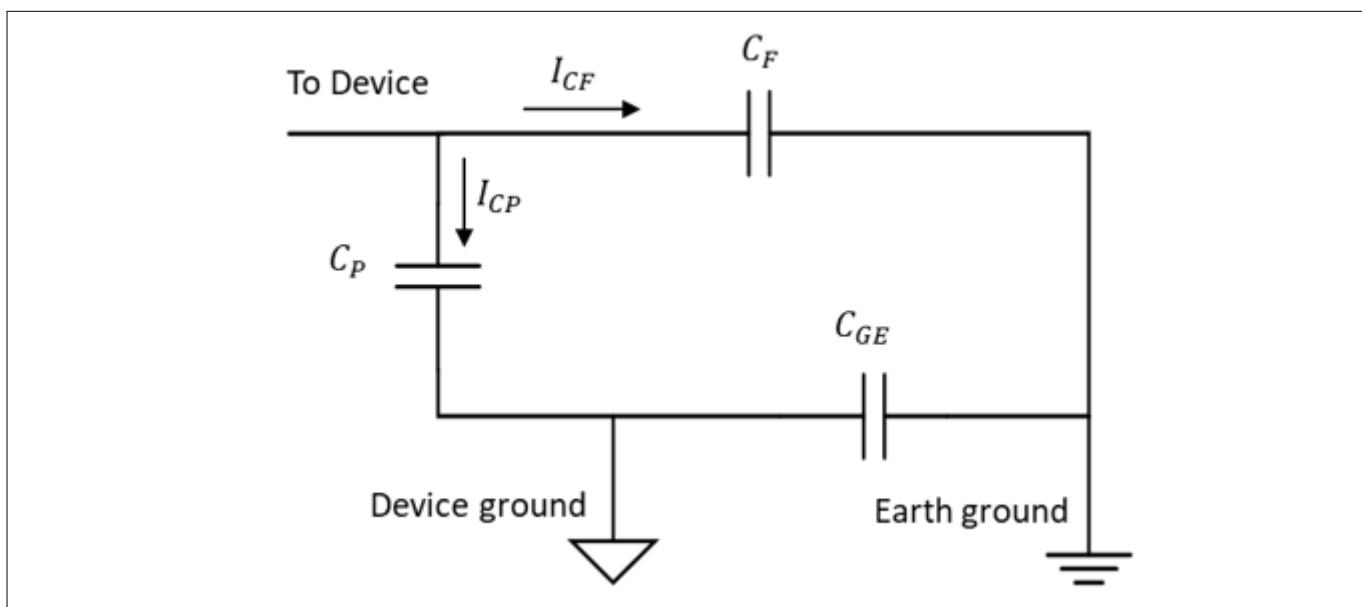


Figure 371 **Equivalent circuit of the CSD for battery-powered application**

Following are the recommendations for a CSD system design in a portable application powered by a battery:

1. Add a large ground plane to the system. The ground plane should be away from the sensing element such that it does not increase the parasitic capacitance of the sensor. Follow the best practices for the [PCB layout guidelines](#) described in this chapter

~~5 PSoC™ 6 application notes~~

- ~~2.~~ Use a driven shield to improve the sensitivity of portable devices. Refer to the [Layout guidelines for shield electrode](#) for more details
- ~~3.~~ Reduce the thickness of the overlay material or use an overlay with better dielectric value to improve sensitivity
- ~~4.~~ Tune the CAPSENSE™ system with powering it by a battery source

5.8.8 CAPSENSE™ Plus

PSoC™ 4 can perform many additional functions along with CAPSENSE™. The wide variety of features offered by this device allows you to integrate various system functions in a single chip, as [Figure 372](#) shows. Such applications are known as CAPSENSE™ Plus applications.

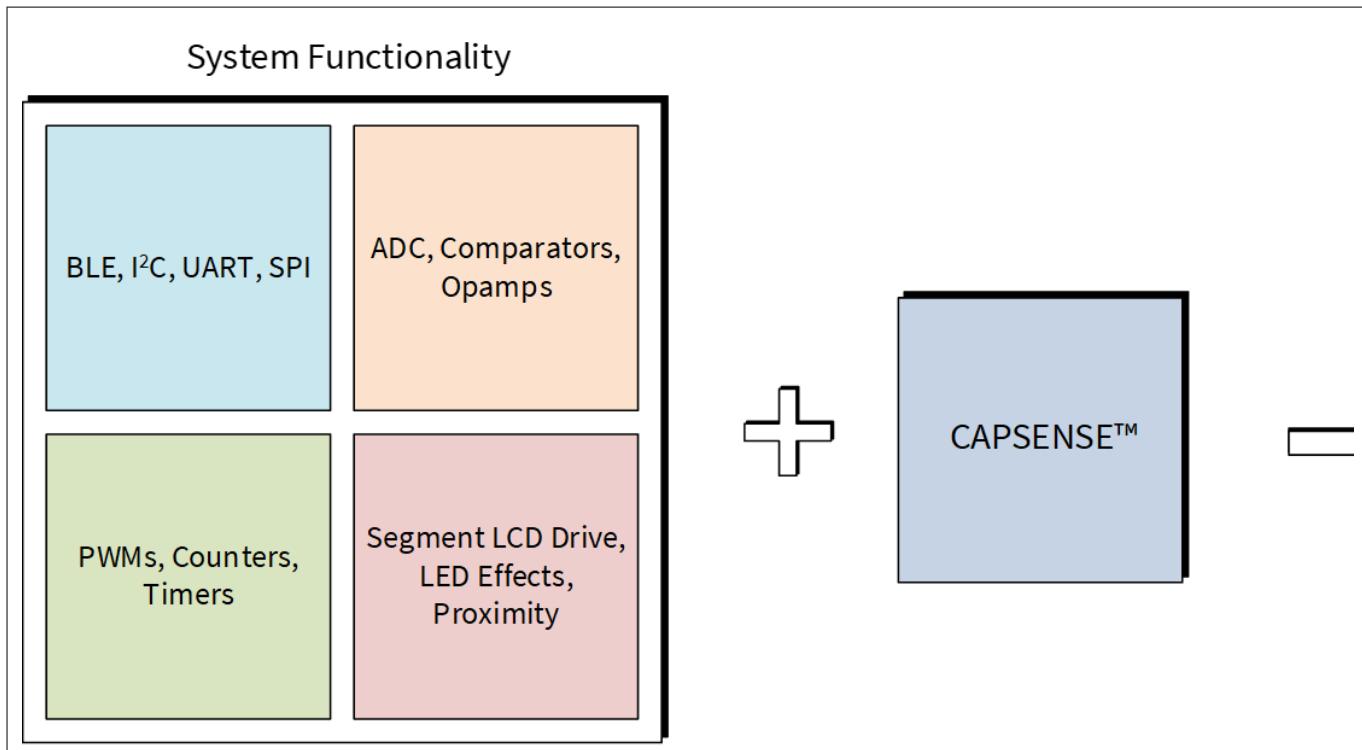


Figure 372 CAPSENSE™ Plus

The additional features available in a PSoC™ 4 device include:

- Communication: Bluetooth® LE, I²C, UART, SPI, CAN, and LIN
- Analog functions: ADC, comparators, and opamps
- Digital functions: PWMs, counters, timers, and UDBs
- Segment LCD drive
- Bootloaders
- Different power modes: Active, sleep, deep sleep, hibernate, and stop

While using above mentioned additional features, it is recommended to configure it in sinking mode as applicable.

For more information on PSoC™ 4, see [AN79953 - Getting started with PSoC™ 4](#), or [AN91267 - Getting started with PSoC™ 4 Bluetooth® LE](#).

The flexibility of the PSoC™ 4 and the unique PSoC™ Creator IDE allow you to quickly make changes to your design, which accelerates time-to-market. Integrating other system functions significantly reduces overall system cost. [Table 77](#) shows a list of example applications, where using CAPSENSE™ Plus can result in significant cost savings.

~~5 PSoC™ 6 application notes~~

~~DRAFT~~ Table 77 Examples of CAPSENSE™ Plus

Application	CAPSENSE™	Opamp	ADC	Comp	PWM, Counter, Timer, UDBs	Comm(Bluetooth® LE, I2C, SPI, UART)	LCD drive	GPIOs
Heart rate monitor (wrist band)	User interface: buttons, linear sliders	TIA, Buffer	Heart Rate Measurement, Battery voltage measurement		LED Driving	Bluetooth® LE	Segment LCD	LED indication
LED bulb	User interface: buttons, radial sliders	Amplifier	LED current measurement	Short circuit protection	LED color control (PrISM*)	Bluetooth® LE		LED indication
Washing machine	User interface: buttons, radial sliders		Temperature sensor	Water level monitor	Buzzer, FOC** motor control	I ² C LCD display, UART network interface	Segment LCD	LED indication
Water heater	User interface: buttons, linear sliders		Temperature sensor, water flux sensor	Water level monitor	Buzzer	I ² C LCD display, UART Network Interface	Segment LCD	LED indication
IR remote controllers	User interface: buttons, linear and radial sliders, touchpads				Manchester encoder			LED indication
Induction cookers	User interface: buttons, linear sliders		Temperature sensor				Segment LCD	LED indication
Motor control systems	User interface: buttons, linear sliders				BLDC*** and FOC motor control			LED indication

(table continues...)

~~DRAFT~~
5 PSoC™ 6 application notes

Table 77 (continued) Examples of CAPSENSE™ Plus

Application	CAPSENSE™	Opamp	ADC	Comp	PWM, Counter, Timer, UDBs	Comm(BLuetoooth® LE, I2C, SPI, UART)	LCD drive	GPIOs
Gaming/simulation controllers	User interface: buttons, touchpads		Reading analog joysticks			I ² C/SPI/UART communication interface	Segment LCD	LED indication
Thermal printers	User interface: buttons		Overheat protection, paper sensor		Stepper motor control	SPI communication interface		LED indication

* PrISM = Precision illumination signal modulation

** FOC = Field oriented control

*** BLDC = Brushless DC motor

Figure 373 shows a general block diagram of a CAPSENSE™ Plus application, such as an induction cooker or a microwave oven.

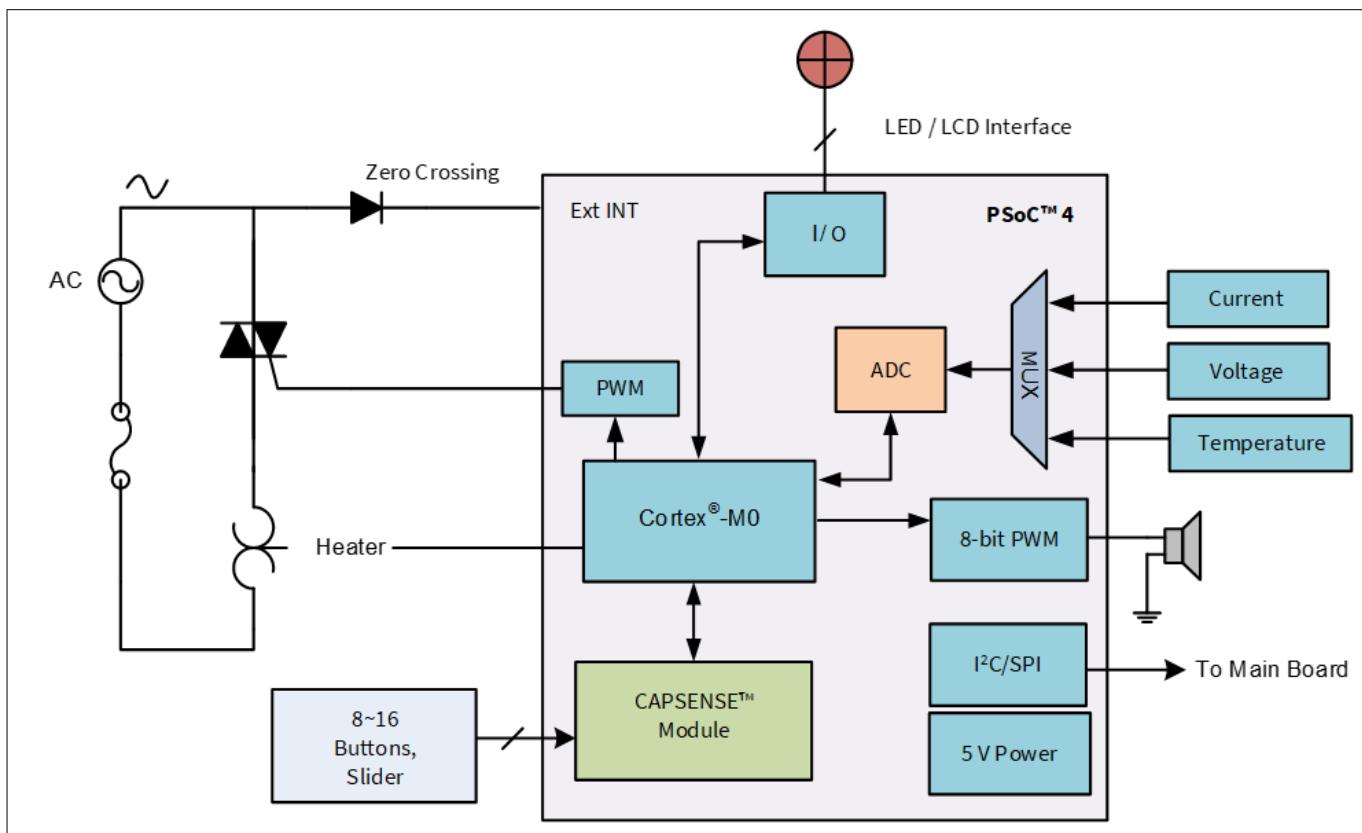


Figure 373 CAPSENSE™ Plus system with PSoC™ 4

In this application, the 12-bit 1 Msps SAR ADC in the PSoC™ 4 detects over-current, overvoltage, and high temperature conditions. The PWM output drives the speaker for status and alarm tones. Another PWM controls the heating element in the system. The CAPSENSE™ buttons and slider constitute the user interface. PSoC™ 4

5 PSoC™ 6 application notes

can also drive a segment LCD for visual outputs. PSoC™ 4 has a serial communication block that can connect to the main board of the system.

~~DETAILED~~ Figure 374 shows the application-level block diagram of a fitness tracker based on PSoC™ 6 MCU with Bluetooth® LE Connectivity. The device provides a one-chip solution and includes features like activity monitoring, environment monitoring, CAPSENSE™ for user interface, Bluetooth® LE connectivity, and so on. For more information on PSoC™ 6 MCU, see [AN210781 – Getting started with PSoC™ 6 MCU with Bluetooth® LE connectivity](#).

5 PSoC™ 6 application notes

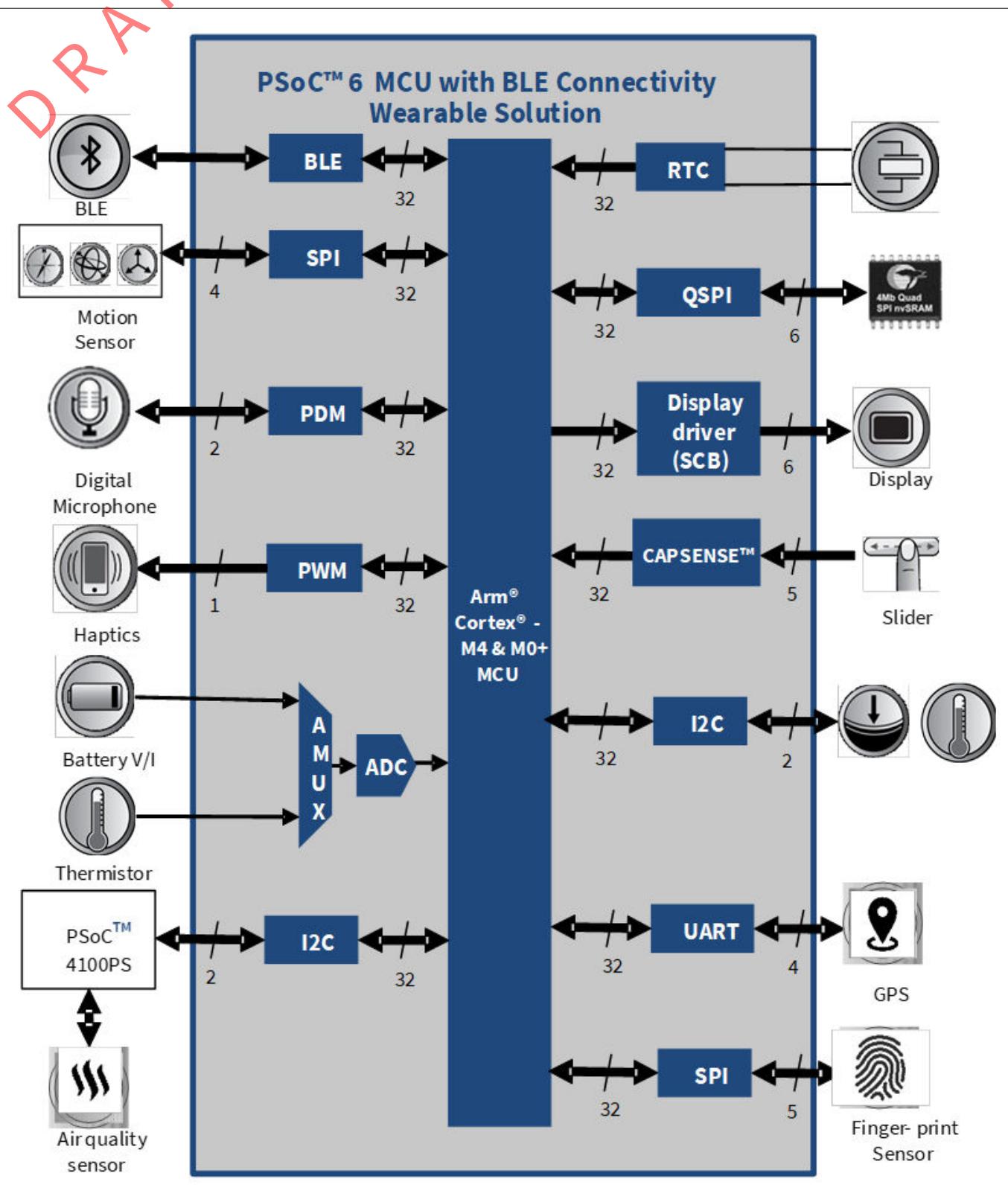


Figure 374

Fitness tracker application with PSoC™ 6 MCU with Bluetooth® LE connectivity block diagram

5 PSoC™ 6 application notes**5.8.9 Resources****5.8.9.1 Website**

Visit the [Getting started with PSoC™ 4](#), [Getting started with PSoC™ 4 Bluetooth® LE](#), [Getting started with PSoC™ 6 MCU](#), and [Getting started with PSoc™ 6 MCU with Bluetooth® LE connectivity](#) website to understand the PSoC™ 4, PSoC™ 6 MCU with Bluetooth® LE connectivity.

5.8.9.2 Device datasheet

- [PSoC™ 4 datasheet](#)
- [PSoC™ 4 Bluetooth® LE datasheet](#)
- [PSoC™ 6 MCU devices](#)

5.8.9.3 Component datasheet/middleware document

- [PSoC™ 4 Capacitive Sensing](#)
- [PSoC™ 6 capacitive sensing](#)
- [CAPSENSE™ middleware library](#)
- [ModusToolbox™ CAPSENSE™ configurator guide](#)

5.8.9.4 Technical reference manual

The [PSoC™ 4 Technical reference manual \(TRM\)](#) and [PSoC™ 6 Technical reference manual \(TRM\)](#) provide quick and easy access to information on PSoC™ 4 and PSoC™ 6 architecture including top-level architectural diagrams, register summaries, and timing diagrams.

5.8.9.5 Development kits

Table 39 lists Infineon® development kits that support PSoC™ 4 and PSoC™ 6 CAPSENSE™.

5.8.9.6 PSoC™ Creator

PSoC™ Creator is a state-of-the-art, easy-to-use integrated development environment. See the [PSoC™ Creator home page](#).

5.8.9.7 ModusToolbox™

ModusToolbox™ software suite is used for the development of PSoC™ 4 and PSoC™ 6 based CAPSENSE™ applications. You can download the ModusToolbox™ software [here](#). The related documents are as follows:

- [ModusToolbox™ release notes](#)
- [ModusToolbox™ install guide](#)
- [ModusToolbox™ user guide](#)
- [ModusToolbox™ quick start guide](#)
- [ModusToolbox™ CAPSENSE™ configurator](#)
- [ModusToolbox™ CAPSENSE™ tuner](#)
- [ModusToolbox™ device configurator](#)

~~5 PSoC™ 6 application notes~~

- ModusToolbox™ SmartIO configurator
- PSoC™ Creator to ModusToolbox™
- ModusToolbox™ command line

5.8.9.8 Application notes

* A large collection of application notes are available to get your design up and running fast. See [PSoC™ 4 application notes](#), [PSoC™ 4 Bluetooth® LE application notes](#), [CAPSENSE™ application notes](#) and [design guides](#).

Following is the list of CAPSENSE™ specific applications notes:

Design guide for PSoC™ 3 and PSoC™ 5LP devices

- [PSoC™ 3 and PSoC™ 5LP CAPSENSE™ design guide](#)

Design guides for the CAPSENSE™ Express family

- [CY8CMBR3XXX CAPSENSE™ design guide](#)
- [CY8CMBR2110 CAPSENSE™ design guide](#)
- [CY8CMBR2016 CAPSENSE™ design guide](#)
- [CY8CMBR2010 CAPSENSE™ design guide](#)
- [CY8CMBR2044 CAPSENSE™ design guide](#)
- [CAPSENSE™ Express™: CY8C201XX application notes](#)

Design guides for PSoC™ 1 devices

- [CY8C20XX7/S design guide](#)
- [CY8C20XX6A/H CAPSENSE™ design guide](#)
- [CY8C21X34/B CAPSENSE™ design guide](#)
- [CY8C20X34 CAPSENSE™ design guide](#)

Getting started application note

- [AN79953 - Getting started with PSoC™ 4](#)
- [AN210781 – Getting started with PSoC™ 6 MCU with Bluetooth® LE connectivity](#)
- [AN221774 – Getting started with PSoC™ 6 MCU](#)

5.8.9.9 Design support

- [Knowledge base articles](#) – Browse technical articles by product family or perform a search on CAPSENSE™ topics
- [White papers](#) – Learn about advanced capacitive-touch interface topics
- [Cypress developer community](#) – Connect with the technical community and exchange information
- [Video library](#) – Quickly get up to speed with tutorial videos
- [Quality and reliability](#) – We are committed to complete customer satisfaction. At our Quality website, you can find reliability and product qualification reports
- <http://www.cypress.com/support> – Submit your design for review by creating a support case. You need to register and login at the website to be able to contact [technical support](#). It is recommended to use PDF prints for the schematic and Gerber files with layer information for the layout

AMUXBUS

Analog multiplexer bus available inside PSoC™ that helps to connect I/O pins with multiple internal analog signals.

5 PSoC™ 6 application notes

~~Baseline~~

A value resulting from a firmware algorithm that estimates a trend in the Raw Count when there is no human finger present on the sensor. The Baseline is less sensitive to sudden changes in the Raw Count and provides a reference point for computing the Difference Count.

~~Button or button widget~~

A widget with an associated sensor that can report the active or inactive state (that is, only two states) of the sensor. For example, it can detect the touch or no-touch state of a finger on the sensor.

Capacitive sensor

A conductor and substrate, such as a copper button on a printed circuit board (PCB), which reacts to a touch or an approaching object with a change in capacitance.

CAPSENSE™

Infineon® Touch-sensing user interface solution. The industry's No. 1 solution in sales by 4x over No. 2.

CAPSENSE™ Mechanical Button Replacement (MBR)

Configurable solution to upgrade mechanical buttons to capacitive buttons, requires minimal engineering effort to configure the sensor parameters and does not require firmware development. These devices include the CY8CMBR3XXX and CY8CMBR2XXX families.

CAPSENSE™ signal

signal (CAPSENSE™ signal)

Difference Count is also called Signal. See Difference Count.

Centroid or Centroid Position

A number indicating the finger position on a slider within the range given by the Slider Resolution. This number is calculated by the CAPSENSE™ centroid calculation algorithm.

CMOD

Modulation capacitor (CMOD)

An external capacitor required for the operation of a CSD block in Self-Capacitance sensing mode.

Compensation IDAC

A programmable constant current source, which is used by CSD to compensate for excess sensor C_P . This IDAC is not controlled by the Sigma-Delta Modulator in the CSD block unlike the Modulation IDAC.

CP

Parasitic capacitance (CP)

Parasitic capacitance is the intrinsic capacitance of the sensor electrode contributed by PCB trace, sensor pad, vias, and air gap. It is unwanted because it reduces the sensitivity of CSD.

CSD

CAPSENSE™ Sigma Delta

CAPSENSE™ Sigma Delta (CSD) is a patented method of performing self-capacitance (also called self-cap) measurements for capacitive sensing applications.

In CSD mode, the sensing system measures the self-capacitance of an electrode, and a change in the self-capacitance is detected to identify the presence or absence of a finger.

5 PSoC™ 6 application notes

~~CSH~~

Shield tank capacitor (CSH)

An optional external capacitor (C_{SH} Tank Capacitor) used to enhance the drive capability of the CSD shield, when there is a large shield layer with high parasitic capacitance.

~~current-output digital-to-analog converter~~

IDAC (current-output digital-to-analog converter)

Programmable constant current source available inside PSoC™, used for CAPSENSE™ and ADC operations.

Debounce

A parameter that defines the number of consecutive scan samples for which the touch should be present for it to become valid. This parameter helps to reject spurious touch signals.

A finger touch is reported only if the Difference Count is greater than Finger Threshold + Hysteresis for a consecutive Debounce number of scan samples.

Difference count

The difference between Raw Count and Baseline. If the difference is negative, or if it is below Noise Threshold, the Difference Count is always set to zero.

Driven-shield

A technique used by CSD for enabling liquid tolerance in which the Shield Electrode is driven by a signal that is equal to the sensor switching signal in phase and amplitude.

Electrode

A conductive material such as a pad or a layer on PCB, ITO, or FPCB. The electrode is connected to a port pin on a CAPSENSE™ device and is used as a CAPSENSE™ sensor or to drive specific signals associated with CAPSENSE™ functionality.

Finger threshold

A parameter used with Hysteresis to determine the state of the sensor. Sensor state is reported ON if the Difference Count is higher than Finger Threshold + Hysteresis, and it is reported OFF if the Difference Count is below Finger Threshold – Hysteresis.

Ganged sensors

The method of connecting multiple sensors together and scanning them as a single sensor. Used for increasing the sensor area for proximity sensing and to reduce power consumption.

To reduce power when the system is in low-power mode, all the sensors can be ganged together and scanned as a single sensor taking less time instead of scanning all the sensors individually. When you touch any of the sensors, the system can transition into active mode where it scans all the sensors individually to detect which sensor is activated.

PSoC™ supports sensor-ganging in firmware, that is, multiple sensors can be connected simultaneously to AMUXBUS for scanning.

Gesture

Gesture is an action, such as swiping and pinch-zoom, performed by the user. CAPSENSE™ has a gesture detection feature that identifies the different gestures based on predefined touch patterns. In the CAPSENSE™ Component, the Gesture feature is supported only by the Touchpad Widget.

5 PSoC™ 6 application notes

~~DRAFT~~ Guard sensor

Copper trace that surrounds all the sensors on the PCB, similar to a button sensor and is used to detect a liquid stream. When the Guard Sensor is triggered, firmware can disable scanning of all other sensors to prevent false touches.

Hatch fill or hatch ground or hatched ground

While designing a PCB for capacitive sensing, a grounded copper plane should be placed surrounding the sensors for good noise immunity. But a solid ground increases the parasitic capacitance of the sensor which is not desired. Therefore, the ground should be filled in a special hatch pattern. A hatch pattern has closely-placed, crisscrossed lines looking like a mesh and the line width and the spacing between two lines determine the fill percentage. In case of liquid tolerance, this hatch fill referred as a shield electrode is driven with a shield signal instead of ground.

Hysteresis

A parameter used to prevent the sensor status output from random toggling due to system noise, used in conjunction with the Finger Threshold to determine the sensor state.

Linear slider

A widget consisting of more than one sensor arranged in a specific linear fashion to detect the physical position (in single axis) of a finger.

Liquid tolerance

The ability of a capacitive sensing system to work reliably in the presence of liquid droplets, streaming liquids or mist.

Low baseline reset

A parameter that represents the maximum number of scan samples where the Raw Count is abnormally below the Negative Noise Threshold. If the Low Baseline Reset value is exceeded, the Baseline is reset to the current Raw Count.

Manual-tuning

The manual process of setting (or tuning) the CAPSENSE™ parameters.

Matrix buttons

A widget consisting of more than two sensors arranged in a matrix fashion, used to detect the presence or absence of a human finger (a touch) on the intersections of vertically and horizontally arranged sensors.

If M is the number of sensors on the horizontal axis and N is the number of sensors on the vertical axis, the Matrix Buttons Widget can monitor a total of $M \times N$ intersections using ONLY $M + N$ port pins.

When using the CSD sensing method (self-capacitance), this Widget can detect a valid touch on only one intersection position at a time.

Modulation IDAC

Modulation IDAC is a programmable constant current source, whose output is controlled (ON/OFF) by the sigma-delta modulator output in a CSD block to maintain the AMUXBUS voltage at V_{REF} . The average current supplied by this IDAC is equal to the average current drawn out by the sensor capacitor.

Modulator clock

A clock source that is used to sample the modulator output from a CSD block during a sensor scan. This clock is also fed to the Raw Count counter. The scan time (excluding pre and post processing times) is given by $(2^N - 1)/\text{Modulator Clock Frequency}$, where N is the Scan Resolution.

5 PSoC™ 6 application notes

~~MSC~~

Multi sense converter (MSC)

The multi sense converter is the analog to digital converter used in Fifth-Generation CAPSENSE™ technology also known as Ratiometric sensing technology.

~~Mutual-capacitance~~

Capacitance associated with an electrode (say Tx) with respect to another electrode (say Rx) is known as mutual-capacitance.

Negative noise threshold

A threshold used to differentiate usual noise from the spurious signals appearing in negative direction. This parameter is used in conjunction with the Low Baseline Reset parameter.

Baseline is updated to track the change in the Raw Count as long as the Raw Count stays within Negative Noise Threshold, that is, the difference between Baseline and Raw count (Baseline – Raw count) is less than Negative Noise Threshold.

Scenarios that may trigger such spurious signals in a negative direction include: a finger on the sensor on power-up, removal of a metal object placed near the sensor, removing a liquid-tolerant CAPSENSE™-enabled product from the water; and other sudden environmental changes.

Noise threshold

A parameter used to differentiate signal from noise for a sensor. If Raw Count – Baseline is greater than Noise Threshold, it indicates a likely valid signal. If the difference is less than Noise Threshold, Raw Count contains nothing but noise.

Noise

CAPSENSE™ noise (Noise)

The variation in the Raw Count when a sensor is in the OFF state (no touch), measured as peak-to-peak counts.

Overlay

A non-conductive material, such as plastic and glass, which covers the capacitive sensors and acts as a touch-surface. The PCB with the sensors is directly placed under the overlay or is connected through springs. The casing for a product often becomes the overlay.

Proximity sensor

A sensor that can detect the presence of nearby objects without any physical contact.

Radial slider

A widget consisting of more than one sensor arranged in a specific circular fashion to detect the physical position of a finger.

Raw count

The unprocessed digital count output of the CAPSENSE™ hardware block that represents the physical capacitance of the sensor.

Refresh interval

The time between two consecutive scans of a sensor.

Scan resolution

Resolution (in bits) of the Raw Count produced by the CSD block.

Scan time

Time taken for completing the scan of a sensor.

5 PSoC™ 6 application notes

~~Self-capacitance~~

The capacitance associated with an electrode with respect to circuit ground.

~~Sense clock~~

A clock source used to implement a switched-capacitor front-end for the CSD sensing method.

~~Sensitivity~~

The change in Raw Count corresponding to the change in sensor capacitance, expressed in counts/pF. Sensitivity of a sensor is dependent on the board layout, overlay properties, sensing method, and tuning parameters.

~~Sensor~~

See [Capacitive sensor](#).

~~Sensor auto reset~~

A setting to prevent a sensor from reporting false touch status indefinitely due to system failure, or when a metal object is continuously present near the sensor.

When Sensor Auto Reset is enabled, the Baseline is always updated even if the Difference Count is greater than the Noise Threshold. This prevents the sensor from reporting the ON status for an indefinite period of time.

When Sensor Auto Reset is disabled, the Baseline is updated only when the Difference Count is less than the Noise Threshold.

~~Sensor ganging~~

See [Ganged sensors](#).

~~Shield electrode~~

Copper fill around sensors to prevent false touches due to the presence of water or other liquids. Shield Electrode is driven by the shield signal output from the CSD block. See [Driven-shield](#).

~~Slider resolution~~

A parameter indicating the total number of finger positions to be resolved on a slider.

~~SmartSense™ auto-tuning~~

A CAPSENSE™ algorithm that automatically sets sensing parameters for optimal performance after the design phase and continuously compensates for system, manufacturing, and environmental changes.

~~SNR~~

Signal-to-noise ratio

The ratio of the sensor signal, when touched, to the noise signal of an untouched sensor.

~~Touchpad~~

A Widget consisting of multiple sensors arranged in a specific horizontal and vertical fashion to detect the X and Y position of a touch.

~~Trackpad~~

See [Touchpad](#).

~~Tuning~~

The process of finding the optimum values for various hardware and software or threshold parameters required for CAPSENSE™ operation.

5 PSoC™ 6 application notes

VREF

Programmable reference voltage block available inside PSoC™ used for CAPSENSE™ and ADC operation.

Widget

A user-interface element in the CAPSENSE™ Component that consists of one sensor or a group of similar sensors. Button, proximity sensor, linear slider, radial slider, matrix buttons, and touchpad are the supported widgets.

~~5 PSoC™ 6 application notes~~

~~DRAFT~~ **5.8.10 Revision history**

Document version	Date of release	Description of changes
2013-04-19	**	New Design Guide.
2013-07-29	*A	Added dual IDAC support. Updated some schematics in chapter 6. Other minor changes to chapters 3, 5, and 6.
2013-11-13	*B	Added support of CY8C4000 devices. Minor fixes throughout the document.
2014-02-24	*C	Updated the table of device features. Changed IDAC names to sync with new PSoC™ Creator Component terms. Added a schematic checklist. Changed screenshots to match the new Component version.
2014-02-27	*D	Updated Table 1-1 per PSoC™ 4000 datasheet.
2014-03-20	*E	Added firmware design considerations to Chapter 6. Added power supply layout and schematic considerations to Chapter 6. Updated the IMO range for PSoC™ 4000
2014-04-15	*F	Updated to support PSoC™ 4000 and PSoC™ Creator 3.0 SP1.
2014-08-29	*G	<p>Added Reference to Getting started with CAPSENSE™ in Proximity (three-dimensional)</p> <p>Renamed Chapter 5.8.2.5 to Liquid tolerance and re-wrote this section.</p> <p>Updated the recommendations for Shield drive that is Csh_tank precharge and C_{MOD} precharge in Chapter 5.8.3.2.7 CAPSENSE™ CSD shielding.</p> <p>Added recommendation for setting “API resolution” in Chapter</p> <p>Added guidelines on how to select value of “Sensitivity” parameter in Chapter</p> <p>Updated recommended values of threshold and hysteresis parameters in Chapter Manual tuning trade-offs</p> <p>Added Section Manual Tuning Slider Example.</p> <p>Updated maximum overlay thickness value for sliders in Table 61.</p> <p>Added guideline on maximum thickness for overlays of materials other than acrylic in Chapter 5.8.7.3.2 Overlay thickness.</p> <p>Re-wrote Chapter Slider design.</p> <p>Added recommendations on DC loads in Chapter 5.8.7.4.5</p> <p>Renamed and rewrote Chapter 5.8.7.4.12 to Layout guidelines for liquid tolerance.</p> <p>Added Chapter External capacitors pin selection.</p> <p>Updated slider related recommendations in Layout rule</p> <p>Updated Electromagnetic compatibility (EMC) considerations, added extensive data on hardware and firmware considerations.</p>
2014-12-19	*H	<p>Added information for the PSoC™ 4 Bluetooth® LE family of devices.</p> <p>Added information for the PSoC Bluetooth® LE family of devices.</p> <p>Updated ground and power layout guidelines in Chapter 5.8.7.4.10 and Chapter 5.8.7.4.11.</p>
2015-01-21	*I	<p>Added information for PSoC™ 4200-M family of devices.</p> <p>Added footnote in chapter Slider.</p> <p>Added GPIO source/sink current limit in Table 68.</p> <p>Changed document title to PSoC™ 4 CAPSENSE™ Design Guide – AN85951</p>

5 PSoC™ 6 application notes

DRAFT

Document version	Date of release	Description of changes
2015-06-02	*J	<p>Changed Document Title to “AN85951 – PSoC™ 4 CAPSENSE™ Design Guide” .</p> <p>Updated Design considerations</p> <p>Updated Preventing ESD discharge.</p> <p>Updated Figure 346.</p> <p>Updated Redirect.</p> <p>Replaced "Guard Ring" with "Ground Ring".</p>
2015-08-20	*K	<p>Added Table 3-1.</p> <p>Removed chapter 3.2.1 C_{MOD} Precharge.</p> <p>Added chapter CAPSENSE™ in PSoC™ 4xxxM/4xxxL-Series.</p> <p>Updated chapter Trace routing.</p> <p>Added reference of AN2397.</p> <p>Added recommendation for modulator clock divider in chapter Manual tuning trade-offs.</p> <p>Added Figure 343</p>
2015-09-16	*L	<p>Updated Chapter 5.8.1.1.</p> <p>Updated 430 Figure 191.</p> <p>Updated Table 39, Table 66, Table 67, Table 69.</p>
2016-01-19	*M	<p>Updated Introduction.</p> <p>Moved Signal-to-noise ratio (SNR) to Chapter 5.8.2.</p> <p>Updated Chapters PSoC™ 4 and PSoC™ 6 MCU and for CAPSENSE™ performance details.</p> <p>Added section to Chapter 5.8.4.</p> <p>Added Glossary.</p>
2016-02-23	*N	<p>Added information on mutual-capacitance sensing in PSoC™ 4 device series.</p> <p>Added information on CAPSENSE™ 3.0 changes.</p> <p>Added following sections:</p> <ul style="list-style-type: none"> • Mutual-capacitance sensing • CAPSENSE™ Architecture in PSoC 4 S-series <p>Updated following sections:</p> <ul style="list-style-type: none"> • Introduction • CAPSENSE™ • CAPSENSE™ design and development tools • CAPSENSE™ performance tuning
2016-03-04	*O	<p>Added PSoC™ Analog Coprocessor references.</p> <p>Updated External capacitors pin selection.</p> <p>Updated Development kits chapter.</p> <p>Updated document title.</p> <p>Updated Copyright notice.</p>
2016-06-14	*P	<p>Updated IDAC sinking mode recommendation.</p> <p>Updated template.</p>
2016-11-18	*Q	Updated Recommended pins for external capacitors .
2017-04-19	*R	Updated logo and copyright.

5 PSoC™ 6 application notes

DRAFT

Document version	Date of release	Description of changes
2017-09-22	*S	<p>Added references to PSoC™ 4100S Plus throughout the document.</p> <p>Updated Chapter 5.8.1.2 CAPSENSE™ features with PSoC™ 4100S Plus features.</p> <p>Updated PSoC™ 4 and PSoC™ 6 CAPSENSE™ development kits with CY8CKIT-149 PSoC™ 4100S Plus prototyping kit.</p> <p>Updated Chapter 5.8.9.8 Application notes with specific list of CAPSENSE™ Application Notes</p>
2018-01-18	*T	<p>Changed document title</p> <p>Added references to PSoC™ 6 MCU features throughout the document</p> <p>Updated CAPSENSE™ generations in PSoC™ 4 and PSoC™ 6 CAPSENSE™ generations in PSoC™ 4 and PSoC™ 6 with generalized architecture block diagram for CSD sensing.</p> <p>Added Gesture in CAPSENSE™ Gesture in CAPSENSE™.</p> <p>Updated Table 39, Table 49, Table 70.</p>
2018-02-28	*U	Added references to PSoC™ 4100PS throughout the document.
2018-11-08	*V	<p>Updated the entire document with references to CY8C62x8 and CY8C62xA devices.</p> <p>Updated the entire document with references to ModusToolbox™.</p> <p>Updated Table 39 with the information of PSoC™ 6 kits.</p> <p>Updated chapter Mutual-capacitance button design with the information of additional mutual cap key.</p> <p>Removed all references to PRoC Bluetooth® LE devices.</p>
2019-04-11	*W	<p>Updated SmartSense and Manual tuning with respect to the latest component.</p> <p>Removed details on different shield drive mode from CAPSENSE™ CSD</p> <p>Updated CAPSENSE™ CSX sensing</p> <ul style="list-style-type: none"> • Updated figures in PSoC™ Creator • SmartSense, and Gesture in CAPSENSE™ with respect to the latest component <p>Removed a table in External capacitors pin selection chapter</p> <p>Updated Table 36</p>
2020-01-07	*X	<p>Added Liquid tolerance for Mutual Capacitance Sensing section</p> <p>Removed Mutual Capacitance Button Design section</p> <p>Updated Table 3-2 and Table 3-3</p> <p>Updated CAPSENSE™ CSX Sensing Method</p> <p>Added ModusToolBox™ section in Chapter 4</p> <p>Updated SmartSense and Manual Tuning section with respect to the latest component.</p> <p>Updated Slider Tuning Guidelines section</p> <p>Added Tuning Shield Electrode section</p> <p>Updated Gesture chapter with gesture tuning guidelines</p> <p>Updated the Low Power design section</p> <p>Updated Sensor and Device placement section</p> <p>Updated Slider Design section</p> <p>Added Effect of Grounding in CSX method and Effect of Grounding in CSD method section</p>

5 PSoC™ 6 application notes

Document version	Date of release	Description of changes
2020-03-18	*Y	<p>Updated Sensor pin selection and chapters.</p>
2020-10-08	*Z	<p>Added CAPSENSE™ configurator CAPSENSE™ configurator.</p> <p>Moved Tuning Shield Electrode section under Chapter 5.8.5.3.2 CSD sensing method (third- and fourth-generation).</p> <p>Added Chapter I am observing a low CM for my CSX button.</p> <p>Added Chapter Mutual-capacitance button design.</p> <p>Added additional layout guidelines in Chapter 5.8.7.4.5 Sensor and device placement.</p> <p>Added additional trace routing guidelines in Chapter 5.8.7.4.7 Trace routing.</p> <p>Added additional guard trace guidelines in Crosstalk Crosstalk.</p> <p>Added new Chapter 5.8.7.5 Noise in CAPSENSE™ system.</p> <p>Added a single line description on self cap buttons in Self-capacitance button Self-capacitance button.</p> <p>Moved ESD and Electromagnetic compatibility (EMC) considerations under Chapter 5.8.7.5.3 External noise.</p> <p>Moved Effect of grounding on CSX method and effect of grounding on CSD method under Effect of grounding.</p>
2021-10-01	AA	<p>Updated to IFX template.</p> <p>Updated CAPSENSE™ features with new Fifth-Generation CAPSENSE™ block and PSoC™ 4100S Max features.</p> <p>Update Table 35.</p> <p>Added new section CAPSENSE™ CSD-RM sensing method (fifth-generation) and CAPSENSE™ CSX-RM sensing method (fifth-generation).</p> <p>Added new section for features Autonomous scanning and Usage of multiple channels.</p> <p>Removed “5.3.2.4 Button widget example” and replaced with Button widget tuning.</p> <p>Added new chapter Touchpad widget tuning.</p> <p>Added new section for Fifth-generation CAPSENSE™ sensing method - CSD-RM sensing method (fifth-generation) and CSX-RM sensing method (Fifth-generation).</p> <p>Updated Table 61, Table 69, and Table 70.</p> <p>Updated Copyright information.</p>
2022-07-21	AB	Template update
2022-12-20	AC	In the Table 35 , in the Feature column, updated "Noise floor (pk-pk)" to "Noise floor (rms)".

5 PSoC™ 6 application notes**5.9 AN219528 PSoC™ 6 MCU low-power modes and power reduction techniques****About this document**

- .
- 9

Scope and purpose

AN219528 describes how to use PSoC™ 6 MCU power modes to optimize power consumption. Major topics include low-power modes in PSoC™ 6 MCU devices, and power management techniques using PSoC™ 6 MCU features. Associated code examples demonstrate different low-power techniques. See [AN230938 – PSoC™ 6 MCU low- power analog](#) for additional information on the low-power analog peripherals of CY8C62x4 family devices.

More code examples? We heard you.

To access an ever-growing list of hundreds of PSoC™ MCU code examples, please visit our [code examples web page](#). You can also explore the video training library [here](#).

5 PSoC™ 6 application notes~~DEAF~~
5.9.1 Introduction

PSoC™ 6 MCU gives the best power-saving benefit when low-power modes are implemented with other power-saving features and techniques, without significantly sacrificing the performance. This application note describes not only general power-saving methods but also the unique low-power modes in PSoC™ 6 MCU. It also discusses other low-power considerations.

This application note requires a basic knowledge of the PSoC™ MCU architecture, and the ability to develop a PSoC™ 6 MCU application using PSoC™ Creator IDE or ModusToolbox™ software. If you are new to PSoC™ 6 MCU, see [AN221774 - Getting started with PSOC™ 6 MCU](#) or [AN210781 – Getting started with PSoC™ 6 MCU with Bluetooth® Low Energy connectivity on PSoC™ Creator](#) IDE. If you are new to PSoC™ Creator IDE, see [PSoC™ Creator IDE](#).

If you are designing an Internet of Things (IoT) system, there are additional connectivity-related power considerations that impact total system power. While these considerations are outside the scope of this application note (AN), there are specific application notes focused on low-power IoT systems. [AN227910 – Low- power system design with AIROC™ CYW43012 Wi-Fi & Bluetooth® combo chip and PSoC™ 6 MCU](#) discusses optimizations for low power in Wi-Fi, Bluetooth®, and PSoC™ 6 MCU systems.

5 PSoC™ 6 application notes

5.9.2 Power modes

5.9.2.1 Power mode transitions

PSoC™ 6 MCU features seven power modes that are divided into system modes that affect the whole device, and standard ARM® CPU modes that affect only one CPU. The system power modes are Low-Power (LP), Ultra-Low-Power (ULP), deep sleep, and hibernate. The ARM® CPU power modes are active, sleep, and deep sleep; these are available in system LP and ULP power modes. [Table 78](#) lists the power modes in which the devices operate.

Table 78 PSoC™ 6 MCU power modes

Power mode	Description
System LP	<ul style="list-style-type: none"> All resources are available with maximum power and speed All CPU power modes supported
System ULP	<ul style="list-style-type: none"> All blocks are available, but the core voltage is lowered resulting in reduced high-frequency clock frequencies All CPU power modes supported
CPU active	<ul style="list-style-type: none"> Normal CPU code execution Available in system LP and ULP power modes
CPU sleep	<ul style="list-style-type: none"> CPU halts code execution Available in system LP and ULP power modes
CPU deep sleep	<ul style="list-style-type: none"> CPU halts code execution Requests system deep sleep entry Available in system LP and ULP power modes
System deep sleep	<ul style="list-style-type: none"> Occurs when all CPUs are in CPU deep sleep CPUs, most peripherals, and high-frequency clocks are OFF Low-frequency clock is ON Low-power analog and some digital peripherals are available for operation and as wakeup sources SRAM is retained
System hibernate	<ul style="list-style-type: none"> CPUs and clocks are OFF GPIO output states are frozen Low-power comparator, RTC alarm, and dedicated WAKEUP pins are available to wake up the system Backup domain is available SRAM is not retained

[Figure 375](#) shows how power mode transitions are based on different events and actions, including interrupts, firmware actions, and reset events. In some cases, mode transitions are done through multiple modes.

For more detailed information, see [PSoC™ 6 MCU architecture technical reference manual](#) and [Appendix A](#).

5 PSoC™ 6 application notes

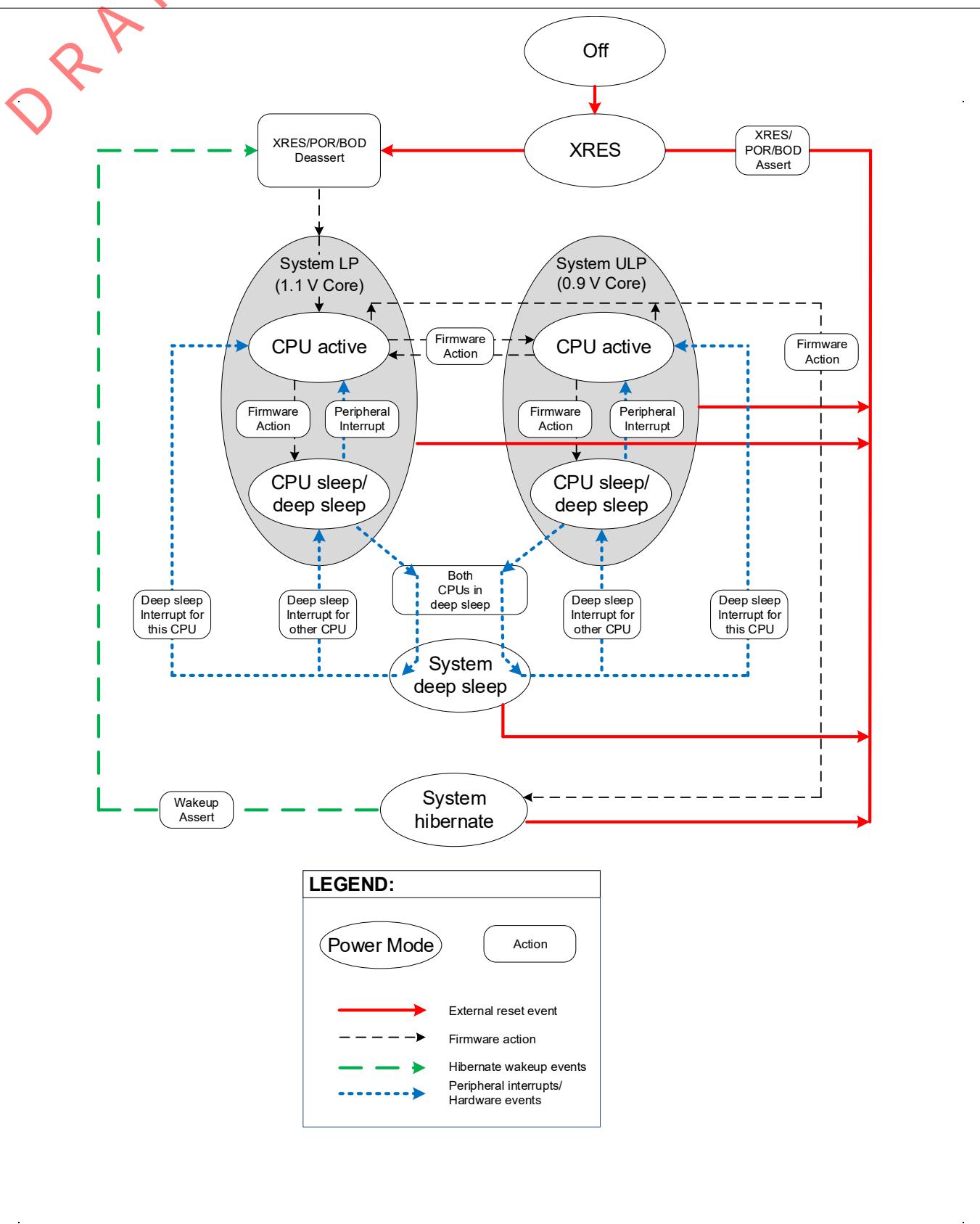


Figure 375

PSoC™ 6 MCU device power mode transition

~~5 PSoC™ 6 application notes~~

~~DRAFT~~ 5.9.2.2 CPU sleep and wakeup instructions

ARM® Cortex® CPUs transition between sleep and wakeup independently. Figure 376 shows several scenarios of wakeup from sleep.

Wait-for-Interrupt (`_WFI`) is the core sleep instruction. After a CPU executes `_WFI`, the CPU goes to sleep and stays in sleep until any interrupt is asserted. Wait-for-Event (`_WFE`) is similar to `_WFI`, but it wakes up when the wakeup event is received instead of an interrupt. Set Event (`_SEV`) is used for waking up other CPUs in sleep mode because of a `_WFE`. CPU deep sleep uses the same instructions for sleep and wakeup, but the SLEEPDEEP bit[2] of the Arm® System Control Register (SCR) is set before a sleep instruction. For more information on SCR, see [ARM® system control register user guide](#). This process is implemented in SysPm PDL library APIs.

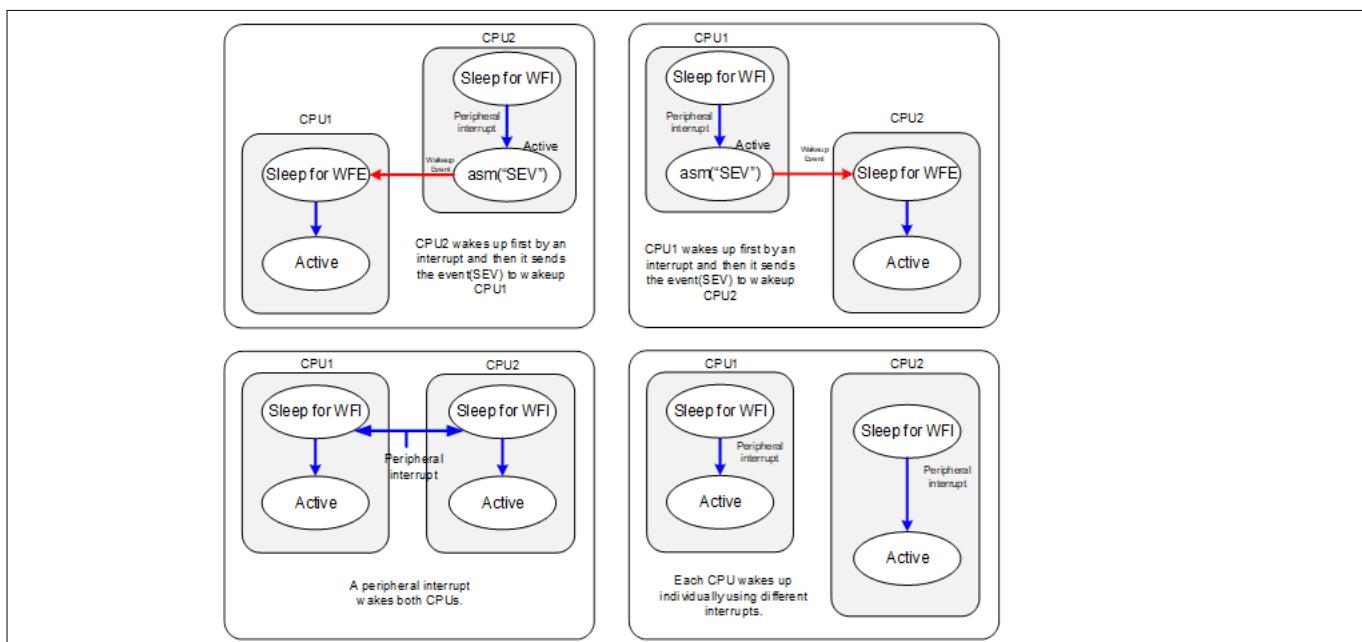


Figure 376 Multi-CPU sleep and wakeup cases

CPU power modes are different from system power modes. Figure 376 shows that each CPU supports its own sleep modes, independent of the state of the other CPU. The device is in system deep sleep mode when both CPUs are in deep sleep. For more detailed information, see [AN215656 – PSoC™ 6 MCU dual-CPU system design](#).

5.9.2.3 Low-power assistant

5.9.2.3.1 Low-power assistant features

The low-power assistant (LPA) provides an easy-to-use GUI for setup of both device and system power options. The LPA's goal is to aid in quickly attaining datasheet power numbers in real-world user applications. For each power option supported in the device, the ModusToolbox™ device configurator has corresponding sections to configure power resources in the same method as configuring any other peripheral.

To launch the assistant, for PSoC™ 6 MCU devices, from under the PSoC™ 6 MCU part number tab and System sub-tab, select Power resource, as shown in Figure 377. For connectivity devices, from under the Wi-Fi or Bluetooth® device part number tab, select the BT or Wi-Fi Resource in the Power section, as shown in Figure 378. See the LPA documentation in the Documentation section located at the top of the Power Configurator.

5 PSoC™ 6 application notes

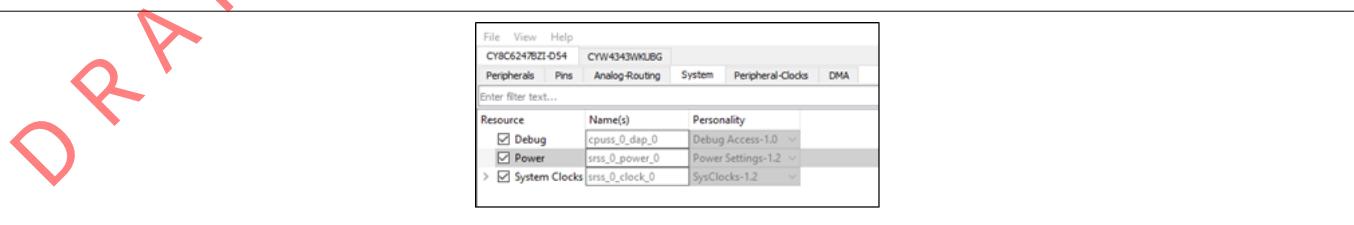


Figure 377 LPA software selection for PSoC™ MCU

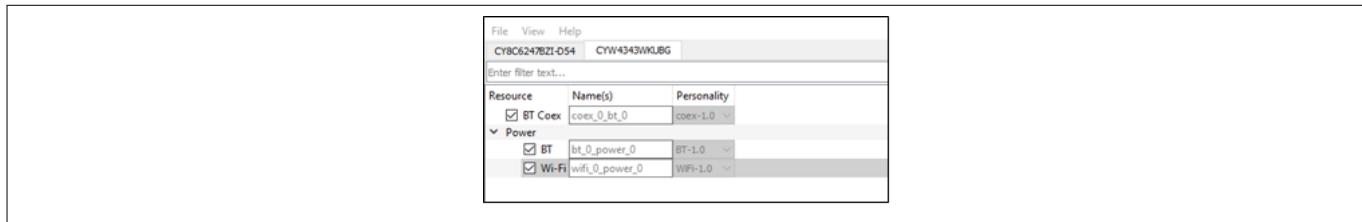


Figure 378 Connectivity device LPA selection

5.9.2.3.2 PSoC™ 6 MCU device LPA settings

Use the PSoC™ 6 MCU device section of the LPA to configure the initial power configuration of the device as shown in [Figure 379](#). Use the General section of the LPA to configure the highest power settings required in the design for system power mode, core regulators, and Vbackup domain features. After the highest power mode is configured, use the RTOS section to configure the lowest power mode the system automatically transitions into. In many designs, this is all the power mode configuration required, because the RTOS typically handles all the dynamic mode transitions. For more advanced use cases, the HAL or SysPm PDL library power functions can be called at any time to dynamically modify any of the power settings or initiate a power mode transition.

CPU and system wakeup from sleep and deep sleep power modes occur when any configured peripheral interrupt is triggered. You can configure the sleep and deep sleep wakeup interrupts by enabling the desired wakeup peripheral's interrupt in that peripheral's configurator within ModusToolbox™ device configurator. If your system enters hibernate mode at runtime using HAL or SysPm calls, the LPA Wakeup pins section simplifies the selection of hibernate wakeup sources.

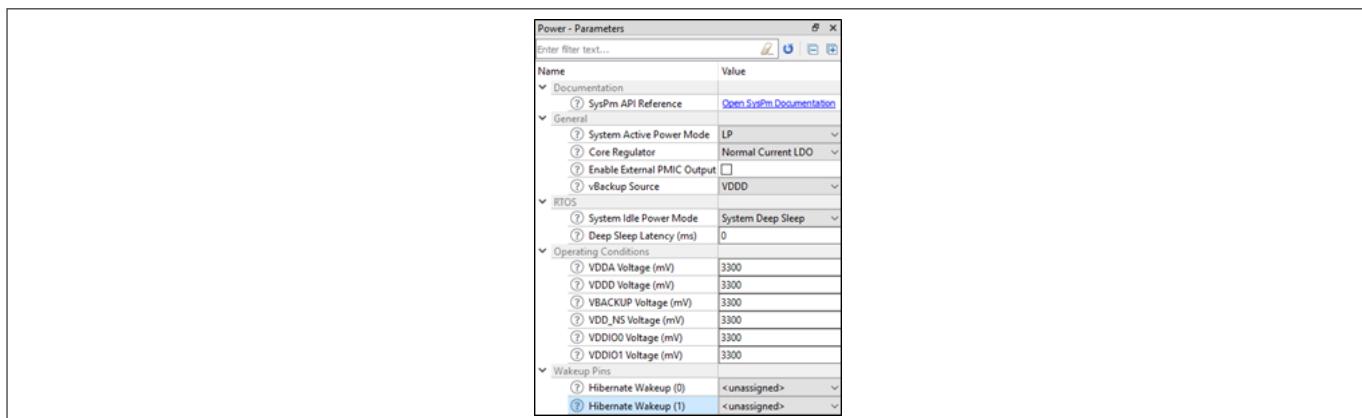


Figure 379 Power parameters: LPA software for PSoC™ MCU

5.9.2.3.3 Connectivity device LPA software settings

Use the “Connectivity device” section of the LPA software to configure Wi-Fi and Bluetooth® power options. AIROC™ Bluetooth® connectivity devices are designed to automatically operate in their lowest power modes. The primary power option is to enable the Bluetooth® device host wakeup trigger as shown in [Figure 380](#).

5 PSoC™ 6 application notes

Bluetooth® host wakeup allows the PSoC™ 6 MCU host device to enter a system deep sleep or hibernate mode and wait for the Bluetooth® connectivity device to wake up the MCU by generating a GPIO pin interrupt when it is required to process Bluetooth® operations.

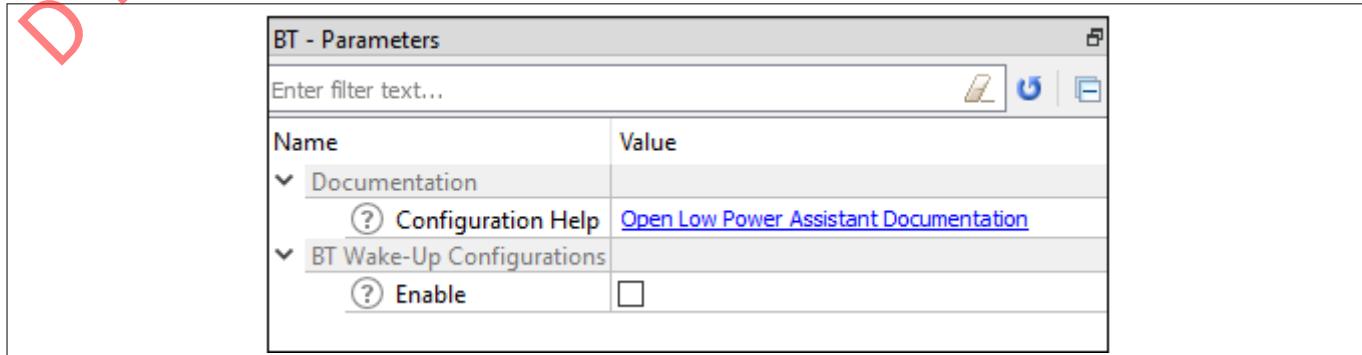


Figure 380 LPA software: Bluetooth® power parameters

Infineon AIROC™ Wi-Fi & Bluetooth® connectivity devices support a wide range of power optimization techniques that you can configure based on the needs of the application and network features supported, as shown in [Figure 381](#). The three primary low-power features for Wi-Fi are host wakeup, offloads, and filters.

- **Host wakeup** allows the PSoC™ 6 MCU host to enter system deep sleep or hibernate mode until the Wi-Fi device requires MCU processing. When the host MCU must wake up, the Wi-Fi device triggers a GPIO pin interrupt. A host wakeup is required to use most of the other low-power Wi-Fi features
- **Offloads** allow the MCU's device stack processing to be moved (offloaded) to the Wi-Fi device internal processor. This increases the time the MCU host can devote to other tasks or spend in a low-power mode
- **Filters** run on the Wi-Fi device and avoid waking the host MCU until the filter conditions are met. An example is an IoT device that needs to respond only to MQTT packets. The device can enable MQTT filters to ignore all other network traffic allowing the host MCU device to stay longer in a low-power mode

5 PSoC™ 6 application notes

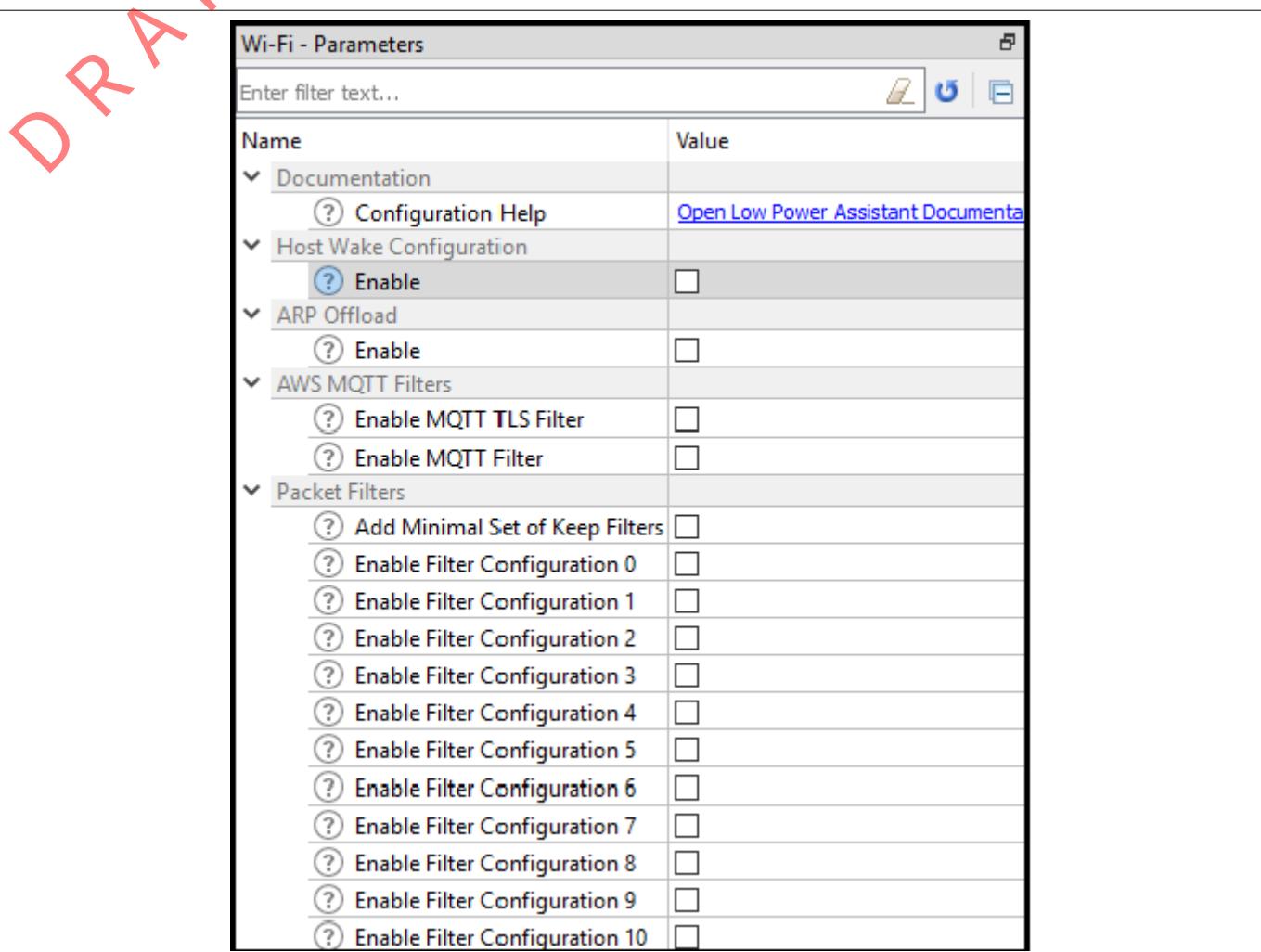


Figure 381 LPA software: Wi-Fi power parameters

5 PSoC™ 6 application notes

5.9.2.4 Subsystem availability and power consumption

5.9.2.4.1 Subsystem availability

Each subsystem resource works differently in various system power modes. For example, the CPU can be in ON, OFF, and Retention modes. It is important to select proper peripherals for the power mode to work correctly. [Table 79](#) lists the resources available in different power modes.

5.9.2.4.2 Approximating power consumption

See the device datasheet for power consumption data for given conditions. Because there are different combinations to achieve the best power consumption, the actual power consumption of the application can be different from the datasheet.

5.9.2.4.3 Power estimator

Eclipse IDE for ModusToolbox™ software version 2.1 provides the Power Estimator (CyPE) tool, which estimates the power consumed by a target device or platform dynamically at runtime. This tool helps you determine the power in different operating modes and how power changes under actual system conditions.



Figure 382

CyPE tool

5 PSoC™ 6 application notes

~~DRAFT~~ 5.9.2.5 Example case scenarios

Proper power mode selection reduces power consumption without performance degradation. [Table 79](#) lists sample scenarios of power modes. In some examples, only a few power modes are used effectively.

Table 79 Sample case scenarios of power modes

Power modes	Wearable device	Air conditioner	Remote controller	Thermometer
System LP CPU active	GUI interaction by user	Motor run	–	Communicates over BLE
System ULP CPU active	Processes heartbeat	–	Sends command	Reads temperature Updates result on LCD
System LP CPU sleep	–	–	–	–
System ULP CPU sleep	Analog block detects heartbeat	–	–	–
System deep sleep	Goes to deep sleep when the device does not detect heartbeat for 30 seconds (device is not in use)	Waits for command No motor run Wakeup by infrared (IR) triggering	–	Wakes up every 1 second using watchdog timer (WDT)
System hibernate	Low battery – Does nothing Resets device when charger is plugged in	–	Waits for button press	–

~~5 PSoC™ 6 application notes~~

5.9.2.6 System power management (SysPm) library

5.9.2.6.1 Overview

The peripheral driver library (PDL) is a complete software tool that includes APIs for configuring peripherals and system registers to implement the desired functionality. PDL provides direct access to almost all hardware resources of the target device. It reduces the need to understand and directly access registers and bit structures.

Within the PDL, the system power management (SysPm) API provides functions to change power modes as shown in [Figure 375](#). The API can also register callback functions to execute a peripheral function before or after power mode transitions as shown in [Figure 383](#). PDL is available from two sources depending on the development platform you are using. The two versions of PDL share the same APIs but are not compatible with the other versions tool base.

- [ModusToolbox™ peripheral driver library](#) (psoc6pdl) from github.com. The ModusToolbox™ software installation automatically installs PDL but requires a download during install directly from GitHub
- [PSoC™ Creator IDE peripheral driver library](#) from the website. The PSoC™Creator IDE download and installation includes PDL so no additional user actions are required

5.9.2.6.2 Mode transition functions

[Figure 375](#) shows firmware transitions for power modes. SysPm provides the default five transition functions for CPU sleep, CPU deep sleep, system hibernate, system LP, and system ULP.

The power mode changing functions provide four different callback operations to execute a necessary action for each peripheral:

Callback function	Description
CY_SYS_PM_CHECK_READY	Checks the ready state to transition to other mode. Exits without transition if it returns CY_SYSPM_FAIL.
CY_SYSPM_BEFORE_TRANSITION	Callbacks execute and configure required actions before mode transition.
CY_SYSPM_AFTER_TRANSITION	Callbacks execute after mode transition or configuration.
CY_SYS_CHECK_FAIL	Callbacks execute only when CY_SYSPM_CHECK_READY fails. It executes the rollback action.

The SysPm driver provides three functions for callback: registration, de-registration, and execution. These functions not only help in power optimization, but also in preventing an abnormal peripheral state after mode transition. The PDL expects the user to register callbacks for each power mode, as shown in [Figure 383](#). Most peripheral drivers have predefined callbacks associated with each power mode. You can choose to register the defined peripheral callback or can make a custom callback. The SysPm transition function executes the registered callbacks sequentially. The first registered function is executed first.

For more information on callback registration and implementation, see [Appendix C](#), and [Appendix D.1](#) of the code example CE219881 - PSoC™ 6 MCU switching between power modes, which is the mode transition example for CPU active, CPU sleep, system LP, system ULP, and system deep sleep.

5 PSoC™ 6 application notes

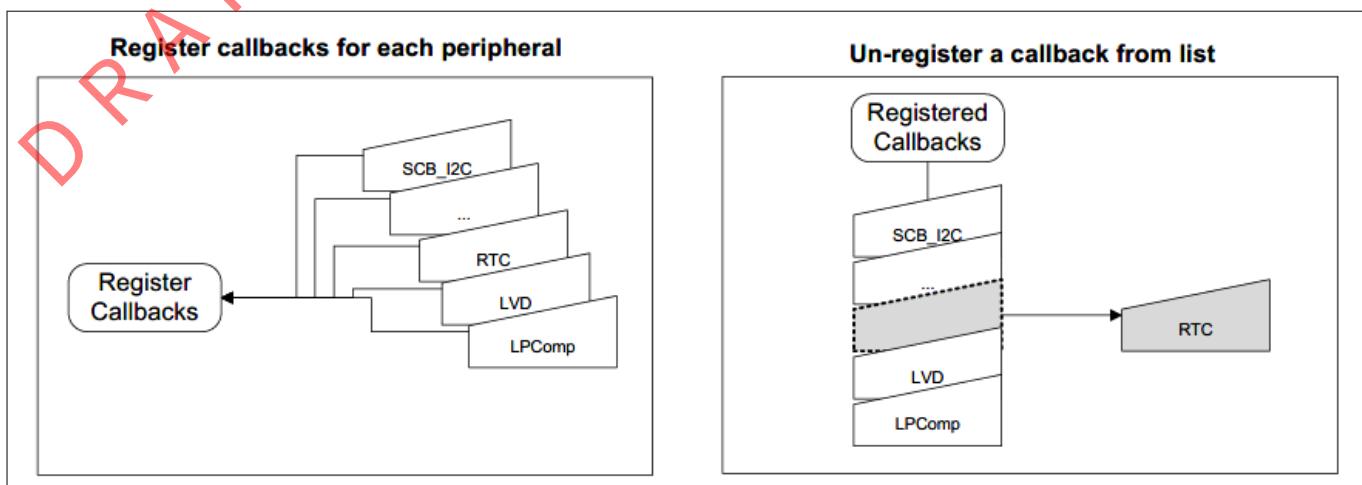


Figure 383 Power mode callback registration and deregistration

By calling the mode transition function, the device starts to transition with four callback operations. The CPU sleep and CPU deep sleep modes use Arm® sleep instructions. Code execution stops and waits for an interrupt during the CPU sleep power mode. Figure 383 shows the CPU waiting for a wakeup source after calling the sleep instruction: __WFI() or __WFE(). After wakeup, the device automatically transitions to CPU active.

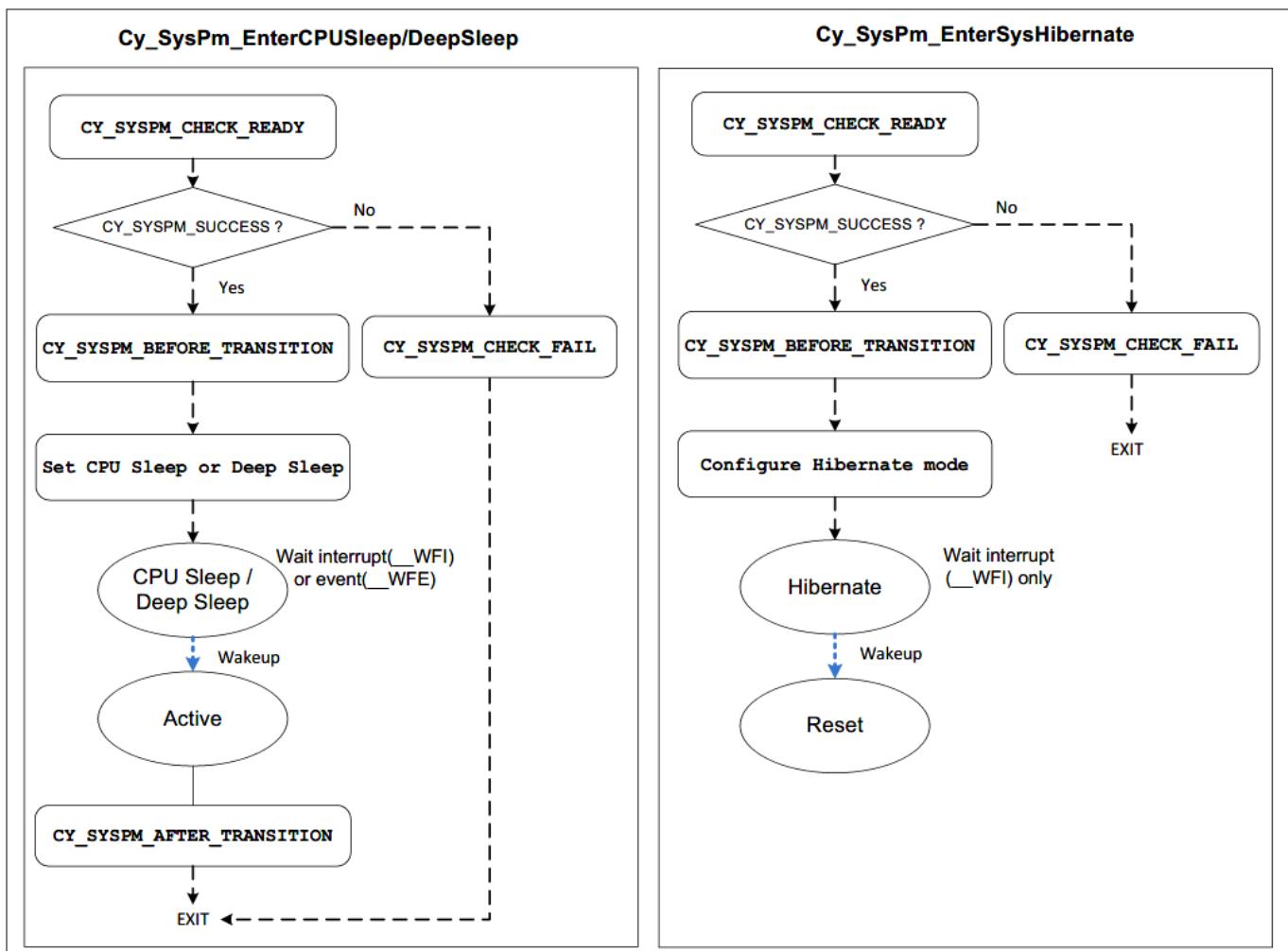


Figure 384 Sleep/deep sleep/hibernate mode transition

5 PSoC™ 6 application notes

In the system LP and system ULP modes, all system resources keep running. Entering system LP and ULP modes is done by configuring the power mode control register; the transition occurs without delay. SysPm PDL provides the associated driver functions, as shown in [Figure 385](#). For the best power efficiency, it is necessary to configure the core voltage regulator and the system clock. For more detailed information, see [Core voltage selection](#), [ULP mode clock](#) and peripheral driver library documentation (PSoC™ Creator > Help > Documentation > Peripheral Driver Library).

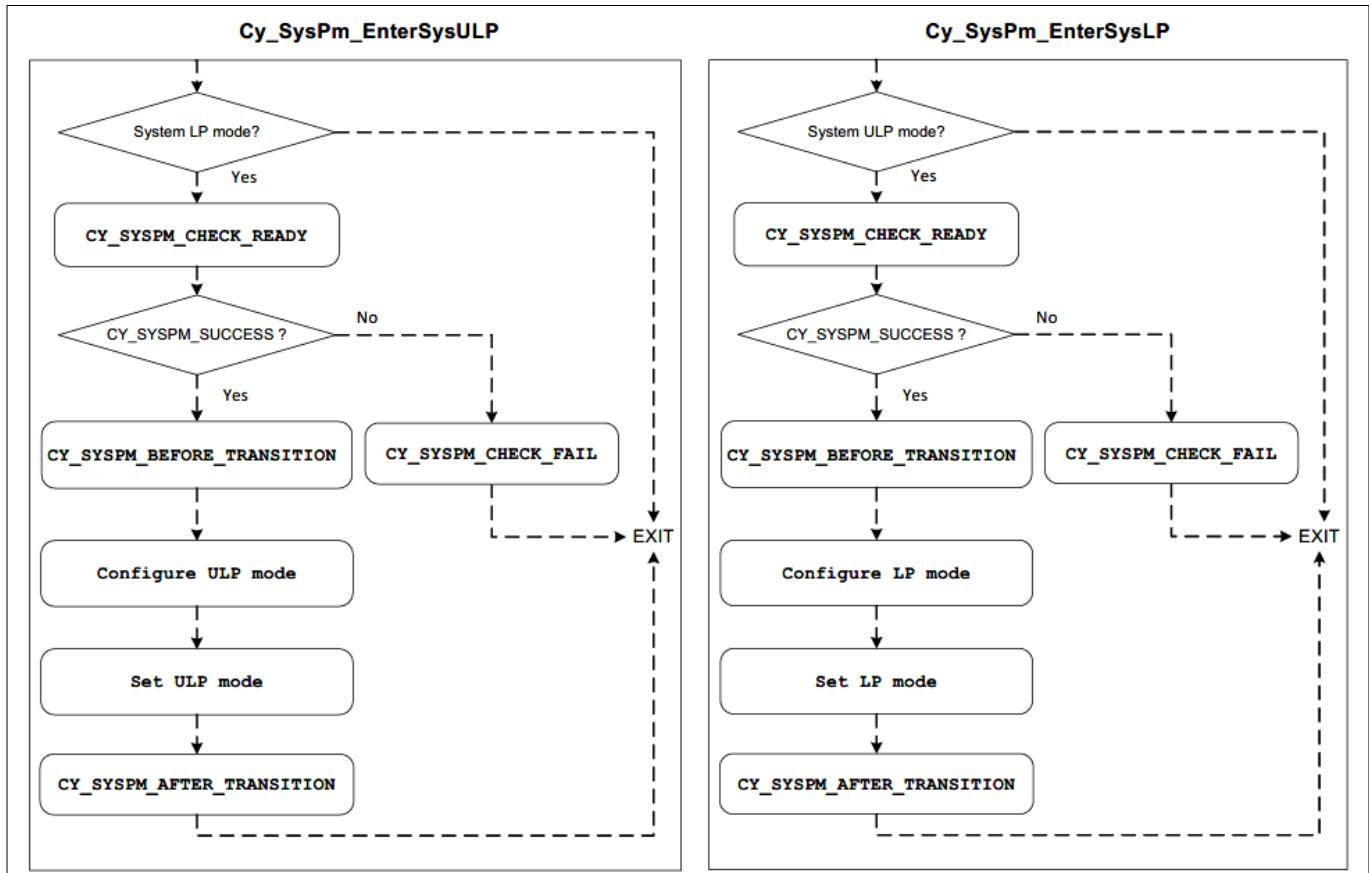


Figure 385 Low-power mode transition

5 PSoC™ 6 application notes

DRAFT

5.9.3 PSoC™ 6 MCU power management

5.9.3.1 Core voltage selection

5.9.3.1.1 Linear regulator and buck regulator

PSoC™ 6 MCU supports multiple on-chip regulators including low drop out (LDO) and single input multiple output (SIMO), or single input single output (SISO) buck to generate VCC for core power, as listed in [Table 80](#). The LDO can provide up to 300 mA in high-current mode (normal) and 25 mA in low-current mode. The buck regulator can provide up to 20 mA for one output and 30 mA combined for both outputs of the SIMO buck. The SIMO buck regulator provides better efficiency under normal load conditions. Once switched to the SIMO buck regulator, it is not possible to switch back to the LDO without resetting the device.

Table 80 Options of core voltage regulators for low-power profile

	Output	Max load	Max clock frequency
LDO	0.9 V	25/300 mA (low-current mode/normal mode)	50 MHz for Arm® Cortex®-M4 (CM4) 25 MHz for Arm® Cortex®-M0+ (CM0+)
	1.1 V	25/300 mA (low-current mode/normal mode)	Allow maximum supportable clock frequency
Buck	0.9 V	20 mA	50 MHz for CM4 25 MHz for CM0+
	1.1 V	20 mA	Allow maximum supportable clock frequency

~~DO NOT USE~~ 5 PSoC™ 6 application notes

5.9.3.2 ULP mode clock

Transition to ULP mode can be done by configuring the power mode control register. There is a maximum clock speed limitation in ULP mode as described earlier, so the clock configuration should be adjusted based on the regulator output when entering or exiting ULP mode. PDL provides associated functions to configure the PWR_CTL register. For more information, see [PSoC™ 6 MCU registers technical reference manual](#).

Figure 386 shows how to transition between LP and ULP modes using PDL functions with the registered callback function. Because of the ULP mode clock limitation, either the frequency-locked loop (FLL) clock speed or HFClk should be adjusted to a valid frequency before the mode transition. Changing the FLL frequency impacts the blocks that use FLL-derived clocks; therefore, all active peripherals should register their own callbacks to handle the changing frequency. [CE219881 – PSoC™ 6 MCU switching between power modes](#) provides an example of clock adjustment using callbacks.

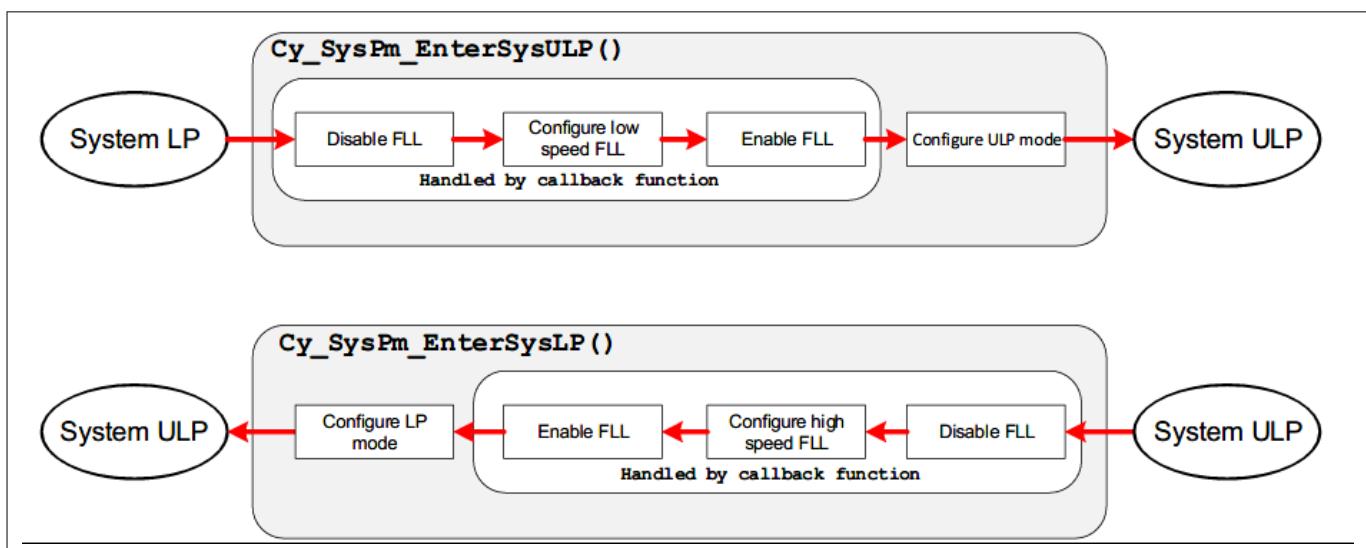


Figure 386 LP mode enter/exit transition

~~DEAFL~~ 5 PSoC™ 6 application notes

5.9.3.3 Backup power domain

The backup domain is not a power mode but rather an always on group of resources that can be active during any of the device power modes.

A separate power supply provides power to a limited number of functions shown in Figure 387 while the main device power is removed. This results in the lowest power state possible while retaining a base level of functionality. The backup domain contains the pmic_wakeup_in pin allowing system-generated signals to trigger the repowering of the device and return to the CPU active power state.

A real time clock (RTC) allows periodic and accurate time-based wakeup triggers. By continuously powering the RTC using the V_{BACKUP} supply accurate time can be maintained independently of the full device power state. Wakeup is accomplished by outputting a HIGH logic level on the pmic_wakeup_out pin most commonly used to enable an external power management integrated circuit (PMIC) supplying power to the full PSoC™ 6 MCU device.

64 bytes of SRAM storage are also maintained allowing key system state information to be retained when the device is repowered. The backup domain is powered by the V_{BACKUP} supply that can be powered by a separate system supply, battery, or supercapacitor. In designs that do not require the backup domain to be powered separately from the rest of the PSoC™ 6 MCU device, V_{BACKUP} should be connected directly to the V_{DDD} supply.

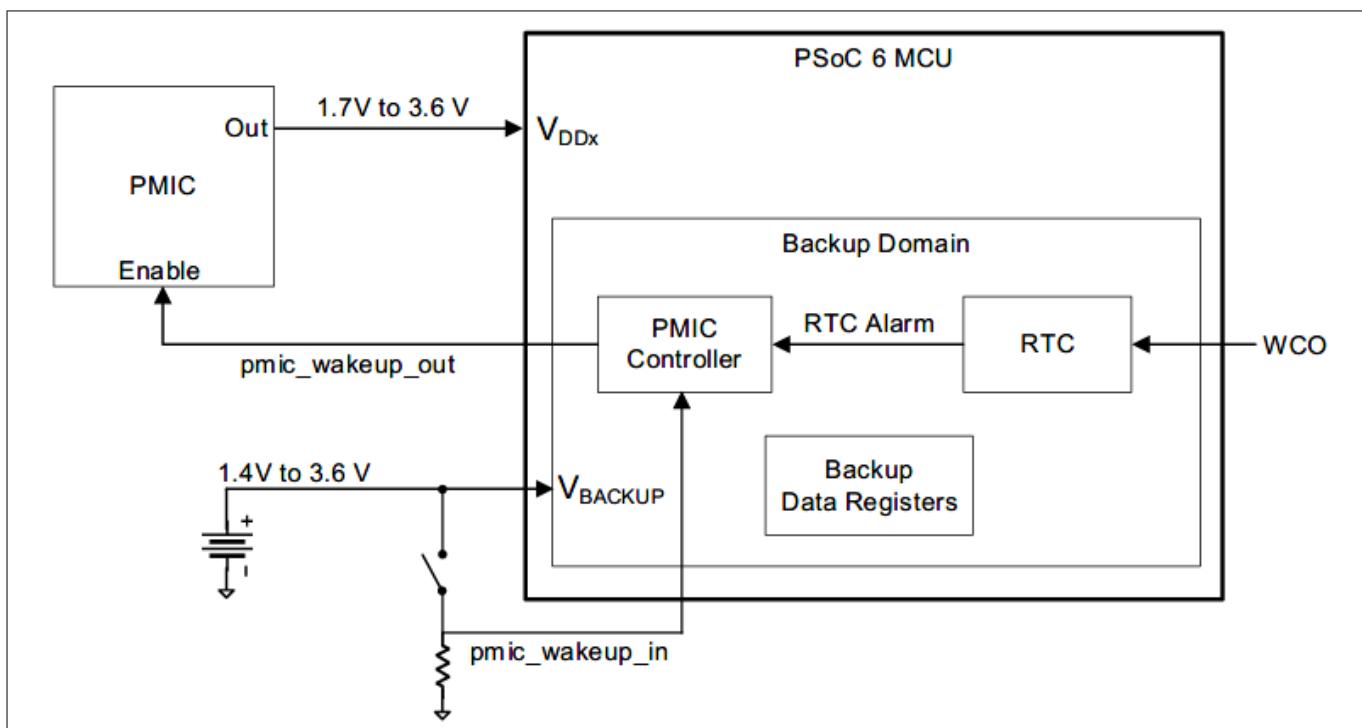


Figure 387 **Backup domain block diagram**

5.9.3.3.1 V_{BACKUP} supply

By providing a separate V_{BACKUP} supply to the backup domain, the majority of the device may be completely powered down eliminating the leakage that would otherwise still occur in system hibernate power mode. If the backup domain is not powered separately from the rest of the device, V_{BACKUP} and V_{DDD} must be tied together.

The internal backup power supply named V_{DDBAK} is automatically switched between V_{DDD} and V_{BACKUP} using an internal circuit. When V_{DDD} is present and greater than 1.6 V, the V_{BACKUP} supply is disabled in order to conserve battery or supercapacitor capacity. As V_{DDD} drops below 1.6 V, the greater of V_{BACKUP} or V_{DDD} supplies is selected and the lower voltage supply connection is disabled. There are no sequencing restrictions between V_{BACKUP} and V_{DDD} . There is no need for external power supply isolation and switching circuits in most designs.

There are four common V_{BACKUP} supply connection options.

5 PSoC™ 6 application notes

- DRAFT**
1. **V_{BACKUP} directly connected to V_{DDD}** – When the backup domain is only used with the full PSoC™ 6 MCU device, there is no reason for a separate supply. V_{BACKUP} and V_{DDD} supply pins must be directly connected together as shown in Figure 388

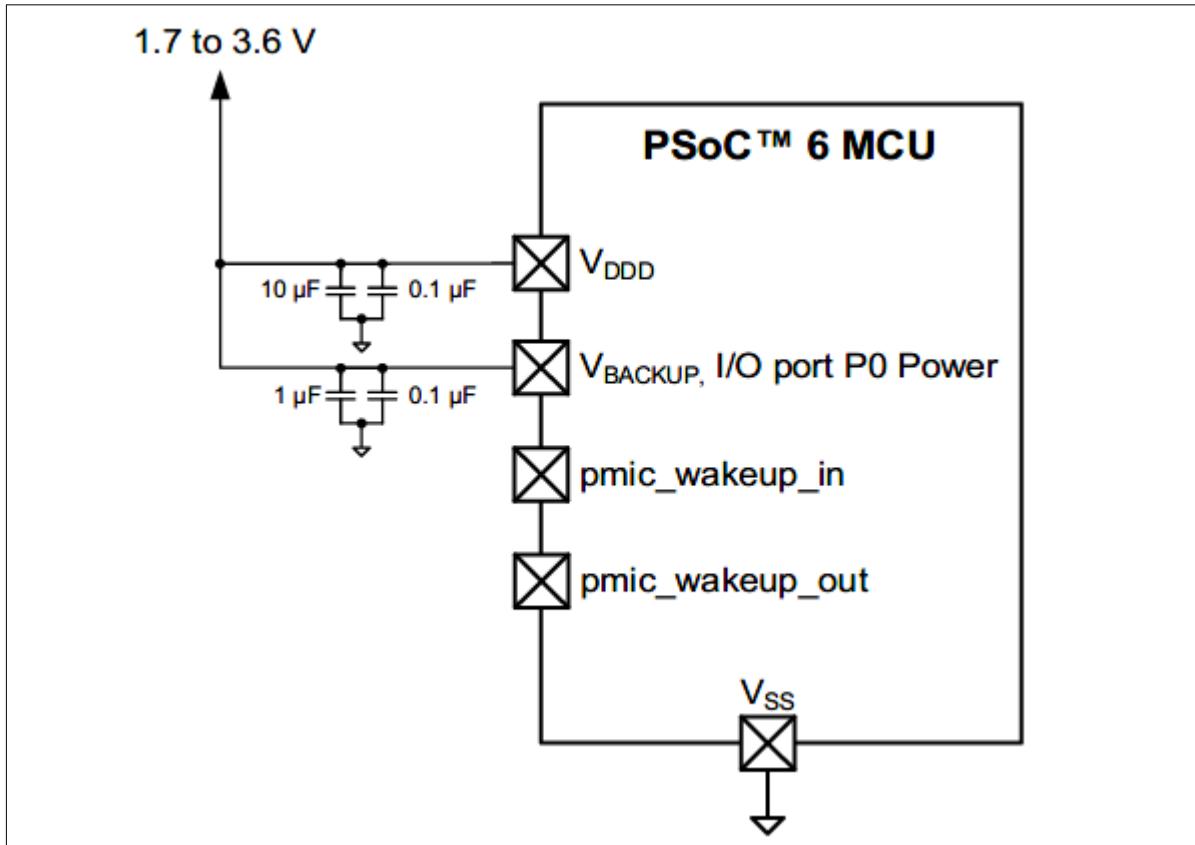


Figure 388 Backup domain combined connection

2. **V_{BACKUP} powered by supercapacitor** – When a supercapacitor is used as the V_{BACKUP} supply as shown in Figure 389, there are two options for recharging it. By calling the `Cy_SysPm_BackupSuperCapCharge()` PDL function, the internal charge circuit is enabled any time the V_{DDD} supply is selected as the active V_{DBBAK} supply

The key value of 0x3C must be passed in the function to avoid accidentally enabling the charge circuit. This charge circuit is not precise; the current varies in the range of 20 μ A based on the difference between V_{DDD} and V_{BACKUP}. Recharge time is on the order of hours

The second option is to use an external charge circuit which can be optimized for any unique design challenges including a faster charge time. Both charge methods can be used in parallel. Any external charge components should be chosen to limit leakage current. The supercapacitor will be charged up to V_{DDD}; therefore, ensure that it has an appropriate voltage rating

5 PSoC™ 6 application notes

DRAFT

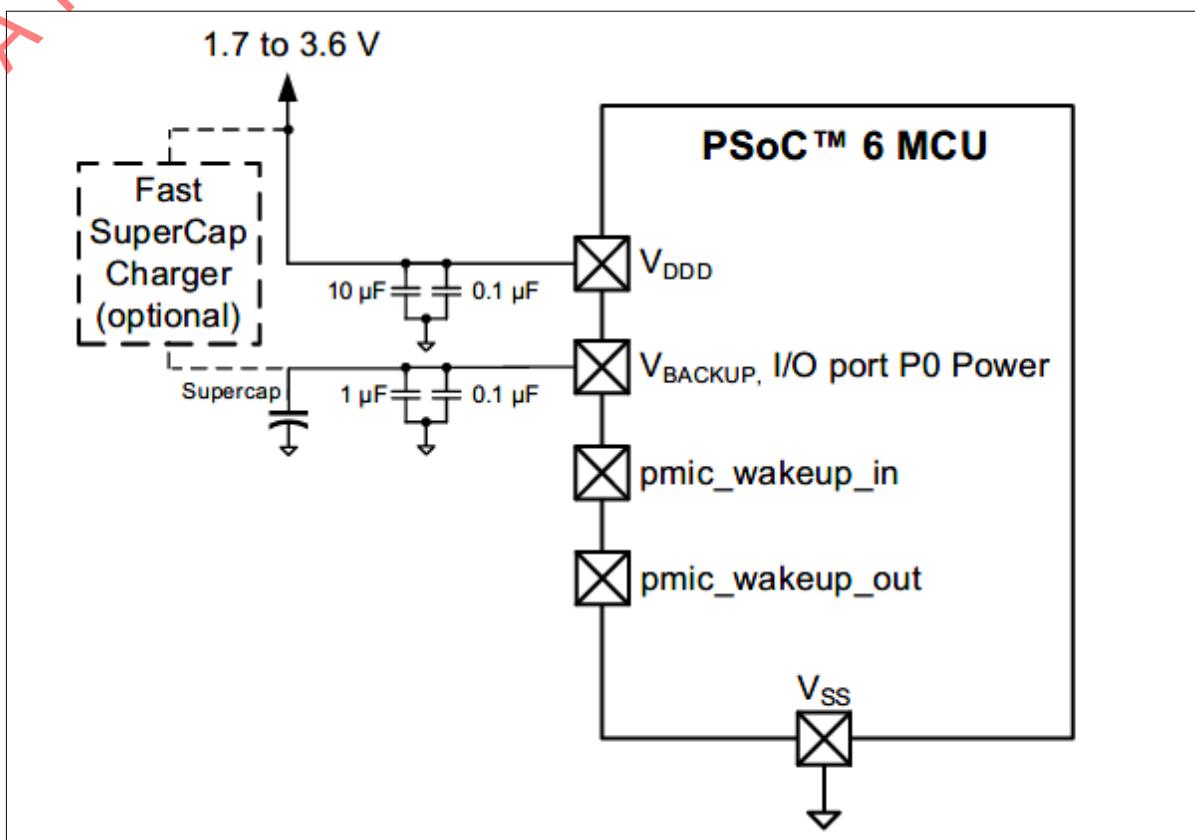


Figure 389

Backup domain supercapacitor connection
Backup domain battery connection

3. **V_{BACKUP} powered by battery** – Figure 390 demonstrates a battery-powered V_{BACKUP} . When a battery supplies V_{BACKUP} , it is important to NOT use the supercapacitor charge circuit because it provides no battery safety mechanisms. If the battery is rechargeable, an external board-level recharge circuit can be provided for the battery-specific charge requirements

5 PSoC™ 6 application notes

DRAFT

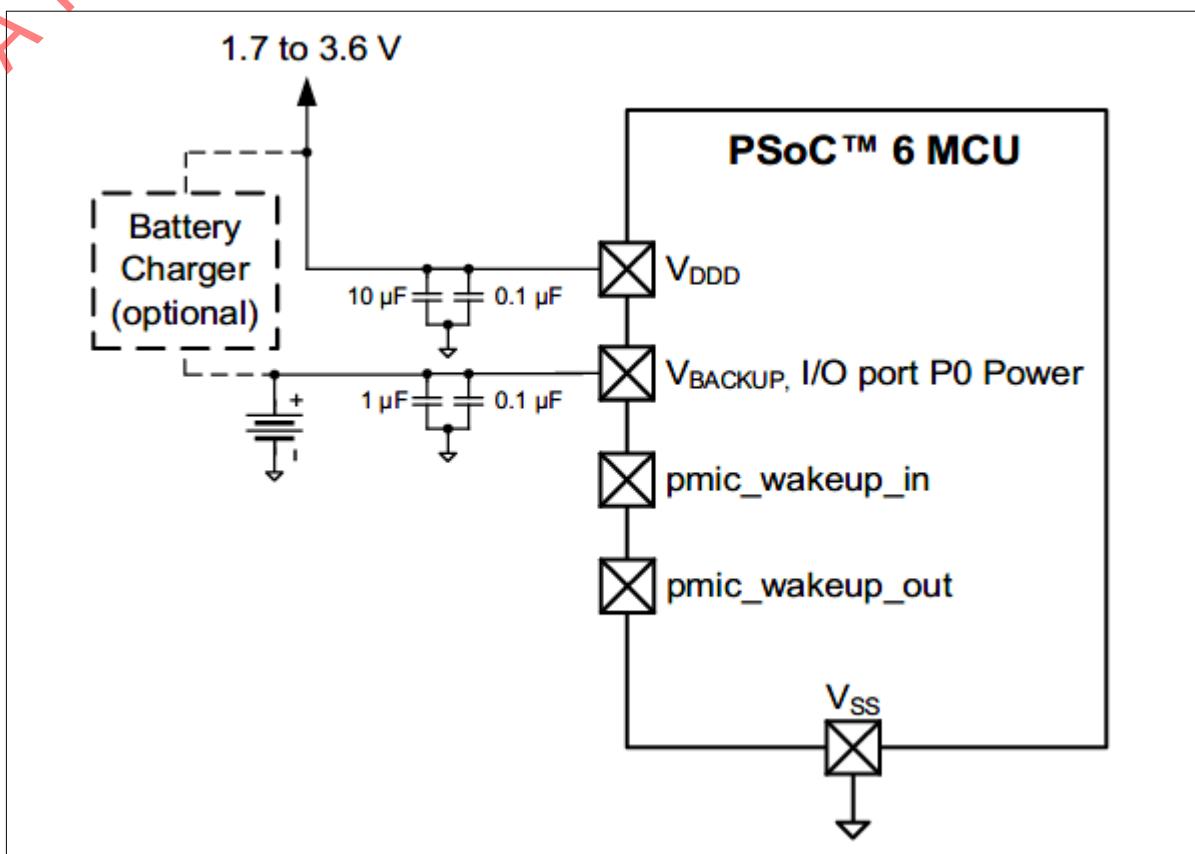


Figure 390 Backup domain battery connection

4. **V_{BACKUP} powered by system provided supply** – If an always-on power supply is already present in the system, it can be used as the V_{BACKUP} supply

On PSoC™ 6 MCU devices with up to 512 KB, 1 MB, or 2 MB of flash, one caution is that the Vmax supply switching logic requires V_{DDD} to be powered up at least once for at least 50 μ s after connecting the V_{BACKUP} supply. This sets the switching circuitry in the correct state to isolate the V_{DDD} supply from the V_{BACKUP} supply by ensuring that there is no leakage path. No code execution is required. If not initialized, the V_{BACKUP} domain will consume significantly more current from the battery and drain it prematurely. Increased leakage does not occur on PSoC™ 6 MCU devices with up to 256 KB of flash and on all newer PSoC™ 6 MCU devices. There are two cases where this leakage issue is likely to occur

1. During manufacture of the PCB, the backup battery is installed and then the device is placed in storage in the high leakage state. Later, when the product is fully assembled, tested, and the PSoC™ 6 MCU device programmed, the logic state will be correctly set but the battery may have drained down during this period
2. If the end product has a user-removable battery and V_{DDD} is powered down, it is important that the PSoC™ 6 MCU be repowered after the user replaces the battery to reset the switch logic to its low-current state

The V_{BACKUP} domain does not contain a low-voltage detect circuit, so it is critical to not let the battery or supercapacitor discharge below 1.4 V. If the V_{BACKUP} supply browns out, a full device power cycle is required to reenable the system. V_{BACKUP} can be directly measured by the ADC by routing a 10% scaled version through AMUXBUSA allowing the system to initiate a supercapacitor fast recharge or notifying the user to change the battery. Measurement is enabled by calling the Cy-SysPm_EnableBackupVMeasure() PDL driver function.

V_{BACKUP} also supplies all of the pins on Port0, so it is important to design their use for minimal current and leakage in your design while powered by the V_{BACKUP} battery or a supercapacitor-based supply.

~~5 PSoC™ 6 application notes~~

~~5.9.3.3.2~~ Backup domain reset

The backup domain is not reset by power-on reset (POR), brownout detect (BOD), watchdog timer (WDT), and external reset (XRES) events as long as V_{BACKUP} or V_{DDD} is present. The backup domain is reset only when both V_{BACKUP} and V_{DDD} are 0 V, or specifically reset through firmware. Under most circumstances, the backup domain should not be reset because it would cause the loss of the RTC time and any data stored in the backup data registers.

Backup domain reset is not required for POR, BOD, XRES, or system hibernate wakeups that occur while powered by the V_{DDD} supply rail. Backup domain reset should occur only when power is removed from the V_{BACKUP} supply or it is allowed to brownout.

The backup power domain is reset under firmware control by calling the `Cy_SysLib_ResetBackupDomain()` PDL function. Firmware must then confirm the `BACKUP_RESET.RESET` register field reads '0' indicating that the reset is completed prior to writing any backup domain registers due to the slower clock rate of the backup domain logic circuits. After the backup domain is reset, the ILO must also be disabled and reenabled to ensure that it is also reset. `Cy_SysClk_IloDisable()` and `Cy_SysClk_IloEnable()` PDL functions perform the ILO reset. The `init_cycfg_platform()` function resets the backup domain and ILO by default when using the initialization code supplied with ModusToolbox™-software.

The steps to reset the backup domain depend on the V_{BACKUP} supply connection.

1. V_{BACKUP} directly connected to V_{DDD} – If V_{BACKUP} and V_{DDD} are connected together, V_{BACKUP} power is removed every power cycle, therefore the backup domain should be reset on all POR and BOD events. If the RTC and backup data registers are not used, it is safe to reset the backup domain after all reset events
2. V_{BACKUP} powered by a supercapacitor – If V_{BACKUP} is connected to a supercapacitor, firmware should measure the supercapacitor voltage immediately after coming out of reset using the ADC. If the voltage on the supercapacitor is less than 1.4 V, it means the RTC and backup data cannot be trusted and firmware should reset the backup domain and allow time for the supercapacitor to recharge
3. V_{BACKUP} powered by battery – The battery voltage should be monitored periodically using the ADC. If it discharges near to its failure point, firmware should set a persistent flag in flash. After booting from reset, firmware can check the flag and know that the RTC and backup data cannot be trusted and the backup domain must be reset. When firmware detects that the battery voltage is good again, it can perform a final backup domain reset and then clear the flag
4. V_{BACKUP} powered by system-provided supply – The system should provide a method of informing the PSoC™ device firmware if the external V_{BACKUP} supply was disabled, allowing firmware to reset the backup domain

5.9.3.3 External PMIC control

The PSoC™ MCU backup domain provides power management IC (PMIC) control shown in [Figure 391](#) to assist in powering up and down the main PSoC™ 6 MCU device supplies (V_{DDD} , V_{DDA} , V_{DDIO}) provided by the PMIC. The `pmic_wakeup_out` pin is designed to connect directly to the PMIC device's enable pin (active HIGH). The `pmic_wakeup_out` pin resets to a logic HIGH, ensuring that the PMIC and full PSoC™ 6 MCU device are initially powered. While fully powered, firmware can configure the `pmic_wakeup_in` pin or RTC wakeup sources and then power down the PMIC and therefore the main device. The PMIC is disabled by driving the `pmic_wakeup_out` pin LOW. When a HIGH logic level is applied to the `pmic_wakeup_in` pin or an RTC alarm occurs, the `pmic_wakeup_out` will be set HIGH, thereby reenabling the PMIC and repowering the main device.

5 PSoC™ 6 application notes

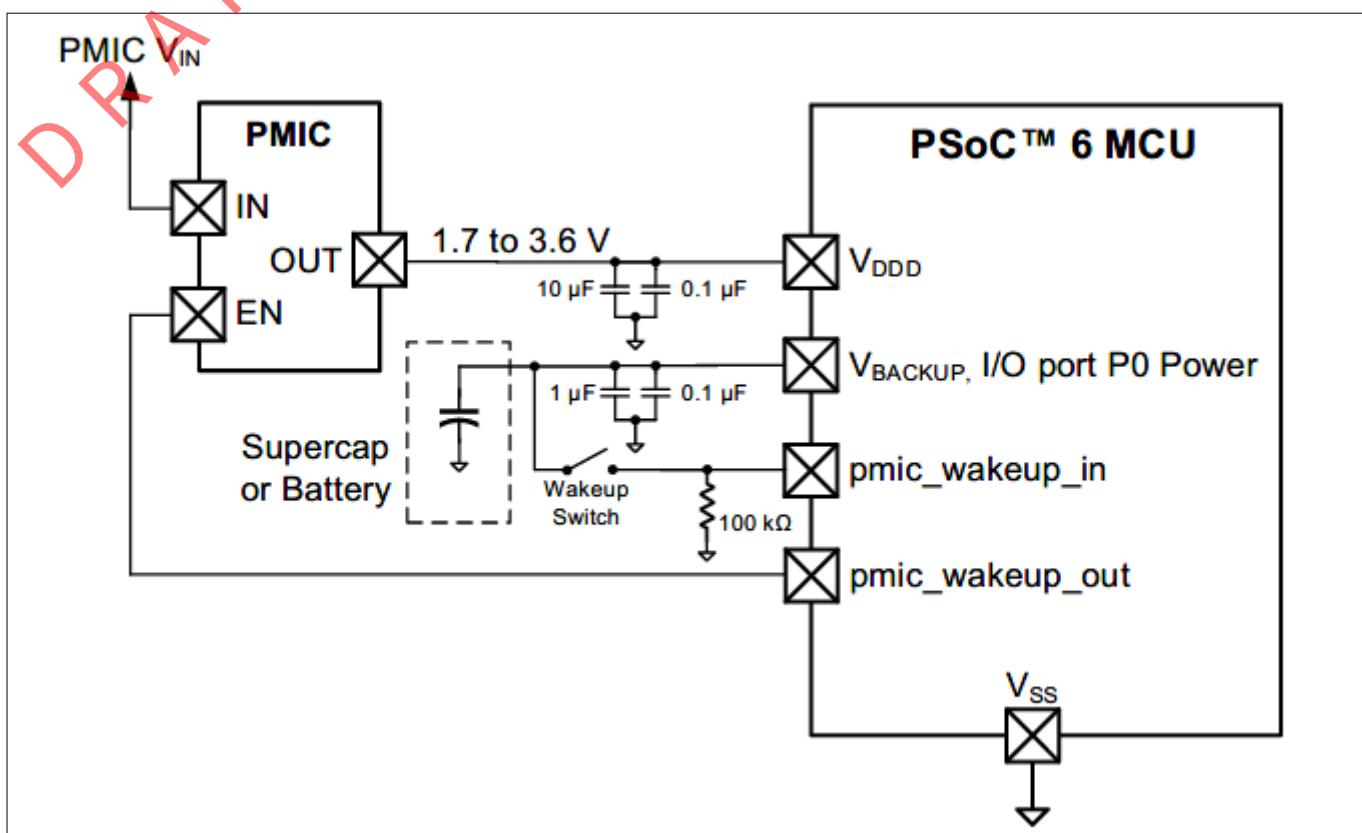


Figure 391 **PMIC controller interface**

To avoid errant code from disabling the PMIC controller, an unlock key is required before any other register writes. A value of 0x3A written to the BACKUP_PMIC_CTL.UNLOCK register disables the PMIC controller and allows the pmic_wakeup_out pin to be used as a GPIO. Any other value written to the UNLOCK field results in the PMIC controller being enabled.

Due to specific register write restrictions, you should use the SysPm PDL library which enforces the restrictions. See the device architecture TRM for full details on direct register access. Enabling the PMIC is a three-step process, which requires an unlock before any other register writes.

```
Cy_SysPm_PmicUnlock();
Cy_SysPm_PmicEnable();
Cy_SysPm_PmicLock();
```

The pmic_wakeup_out pin can be permanently set HIGH by executing `Cy_SysPm_PmicAlwaysEnable()`. After calling this function, the PMIC output cannot be disabled until V_{BACKUP} power is removed or the backup domain is reset.

Setting the pmic_wakeup_out pin LOW is accomplished by clearing the BACKUP_PMIC_CTL.PMIC_EN register bit-field. Disabling the PMIC using PDL requires two PDL instructions. The PMIC controller should not be relocked after disabling using PDL because this will automatically re-enable the PMIC. The length of time from calling `Cy_SysPm_PmicDisable()` until V_{DDD} decays is dependent on the external power supply design.

```
Cy_SysPm_PmicUnlock();
Cy_SysPm_PmicDisable();
```

~~5 PSoC™ 6 application notes~~

There are many options other than a direct connection to a dedicated PMIC. This flexibility allows optimizing power switching to the needs of the design. Some of the common pmic_wakeup_out pin connections are as follows:

1. Advanced PMIC device providing multiple supply rails to V_{DDD} , V_{DDA} , and V_{DDIO}
2. Simple LDO or switching regulator with enable pin supplying all PSoC™ 6 MCU device power pins
3. Gate of low-leakage N-FET or equivalent switch allowing control of an existing regulated supply or a direct connection to a battery
4. Digital signal to an external MCU or a controller managing the system power

5.9.3.3.4 Wakeup sources

The pmic_wakeup_in pin is the most flexible and lowest-current method of waking and repowering the PSoC™ 6 MCU because any circuit that can produce a HIGH logic level can be used. One of the most common methods is to connect the pin to a pull-down resistor and the user “power” button tied HIGH. If multiple wakeup sources are desired, a wired OR circuit can trigger the wake pin. The device can then optionally query the source of wakeup by using additional pins after power up. [Figure 392](#) demonstrates a wired OR wakeup for detecting the press of the product’s power button or insertion of the USB cable.

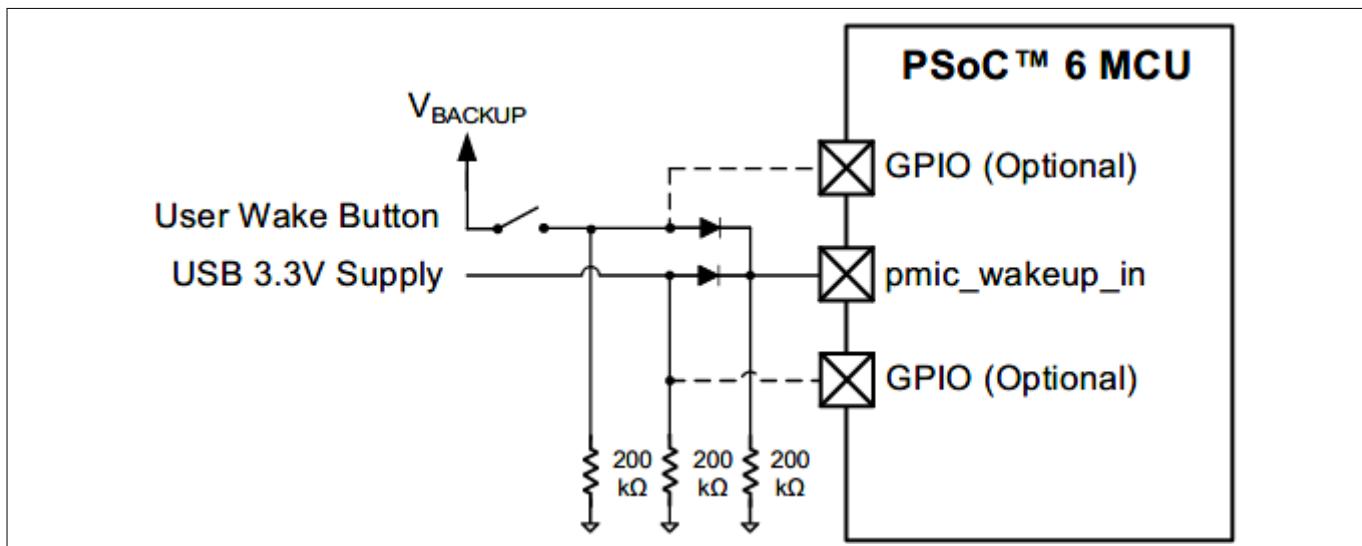


Figure 392 Multiple PMIC wakeup inputs

The pmic_wakeup_in and hibernate_wakeup[1] signals share the same physical pin on Port0 and therefore have a restriction if both system hibernate and PMIC control are used in a design. The hibernate_wakeup[1] pin can wake the device on either a LOW or HIGH logic level that is set by its register configuration. The pmic_wakeup_in pin is active only with a HIGH logic level. If using both pin signals in the same design, ensure that a HIGH logic level is used to wake from hibernate and wake the PMIC. Another option is available on devices with a second hibernate_wakeup[0] pin, allowing the backup domain wakeup signals to be separated from hibernate wakeup on the second pin.

The RTC should be used for periodic wakeup of the full device if required because the WDT is not available in the backup domain. Use of the RTC requires a WCO be installed, thereby increasing the component count and current consumption. Wakeup using the RTC can be triggered at a specific alarm time using two separate alarms or every rollover of a periodic time based on day, hour, minute, or second. The RTC also allows external clock inputs from external sources like watch crystal oscillators and 50/60 Hz mains power. To aid in keeping an accurate time, a calibration register allows +/-1 ppm trim resolution of the WCO frequency. The RTC has specific data access restrictions; therefore, you should use the Cy_RTC PDL library which handles them properly. See the device architecture TRM for full details on direct register access to the RTC.

5 PSoC™ 6 application notes~~DRAFT~~ **5.9.3.3.5 Backup data registers**

The backup domain includes sixteen 32-bit registers, BACKUP_BREG[15:0], that retain their contents as long as the V_{BACKUP} supply is valid. Each register holds 4 bytes of data for a total of 64 bytes. These registers are used to retain important system information and flags during power down of the full device. They can also be used during system hibernate mode allowing data to be read after device wake and reset.

DRAFT

5 PSoC™ 6 application notes

5.9.4 Other power saving techniques

5.9.4.1 Use PSoC™ 6 MCU to gate current paths

Your PCB may contain other components that draw power; PSoC™ 6 MCUs can be used to control the power through them by supplying the power with GPIO pins that can be turned ON and OFF in firmware. Note that the maximum pin source and sink capabilities listed in the datasheet must not be exceeded. If a higher current is required than the GPIO can directly supply, use external power devices that are controlled by the GPIO.

A good example of this scenario is a low-power comparator (LPComp) application as shown in [Figure 393](#). In this case, the PSoC™ device compares the voltage on an analog pin, which changes as the potentiometer resistance changes.

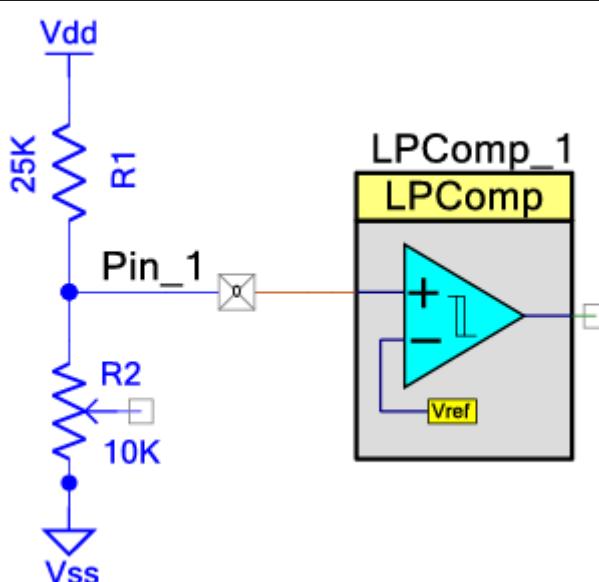


Figure 393 Typical LPComp application

LPComp can be turned OFF when not in use, but external components will still consume power because the current path through the resistor and potentiometer remains. A simple solution with PSoC™ 6 MCU is to use a second pin as a switch to ground, as shown in [Figure 394](#).

In this configuration, the current flow can be stopped by writing a '1' to Pin_3 and allowing the pin to float. This removes the current consumption by reducing the voltage differential across the two resistors to 0 V. Writing a '0' resumes the current flow. Only one pin and a few lines of code are required to implement this power-saving feature.

5 PSoC™ 6 application notes

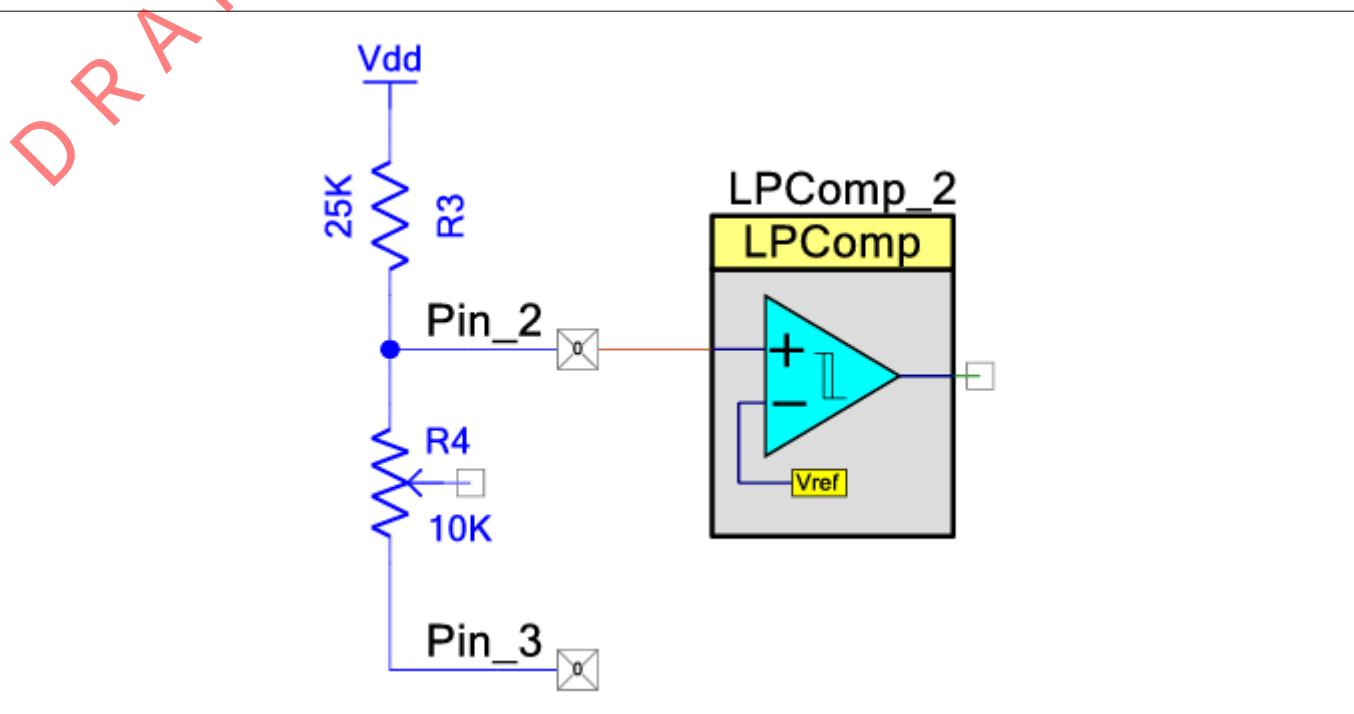


Figure 394 Using a GPIO as a ground switch

5.9.4.2 Disable unused blocks

You can save unnecessary current consumption by disabling unused blocks. The power saved depends on the block disabled.

5.9.4.3 Use DMA to move data

You can save power any time you offload a task from the CPU and either halt the CPU or let it do something else in parallel. The DMA engine can be used in both system LP and ULP modes to transfer data with no CPU use. The power saved is either the difference between CPU active and CPU stop power modes if the CPU can be halted, or a lower CPU active current if the CPU can be clocked at a slower frequency and still get the same work done.

5.9.4.4 Periodic wakeup timers

A periodic wakeup from CPU sleep mode is the most common way to reduce power consumption. The average power consumption is determined by the ratio of CPU active period power consumption and CPU sleep period power consumption. To achieve the best result, the sleep period should be as long as possible and the active period should be as short as possible. An example of CPU sleep mode power savings from the CY8C61x6 datasheet is CM4 active = 1.7 mA (SIDF5) and CM4 sleep = 0.76 mA (SIDS7).

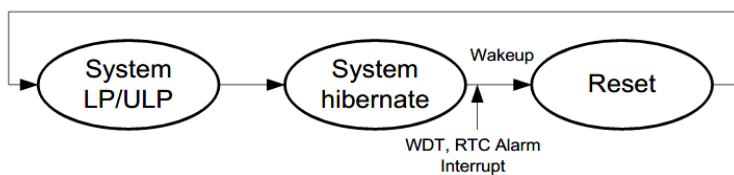
The WDT and multi-counter WDT (MCWDT) are effective periodic wakeup sources in system deep sleep and system hibernate modes. If your application needs longer, or more precise wakeup periods, an RTC alarm can be a good periodic wakeup source. See [Appendix D.3](#) for a code example using the RTC periodic timer for wakeup. An example of system deep sleep mode power savings from the CY8C61x6 datasheet is CM4 active = 1.7 mA (SIDF5) and system deep sleep = 7 μA (SIDD5).

The following scenarios show how to change the mode states:

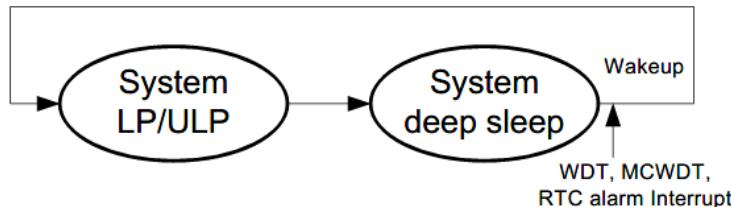
- System hibernate

5 PSoC™ 6 application notes

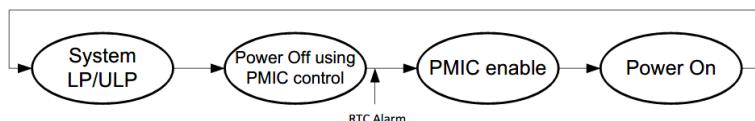
DRAFT



- System deep sleep



- Power OFF with an external PMIC



5.9.4.5 Disabling CPUs

If the CM4 CPU is not used in the application, disable it by calling the `cy_SysDisableCM4()` function from the CM0+ firmware. You can also set the `Clk_Fast` divider to 256 to minimize any fan-out leakage from this clock. If the CM0+ CPU is not used in the application, place it into system deep sleep power mode and set the `Clk_Slow` divider to 256 to minimize any fan-out leakage from this clock.

Note: `Clk_Slow` also affects how fast the DMA engine runs.

5.9.4.6 Splitting tasks between the CPUs

There are several reasons to choose either CM4 or CM0+ to do certain tasks. For example:

1. Tasks that require floating point or DSP operations are more efficiently run on CM4
2. Tasks that are time-critical requiring the maximum clock speed (150 MHz) should run only on CM4
3. If deterministic timing is required by a single task independent of all other tasks, it is often beneficial to move that task to its own CPU to simplify timing. Usually, the CM0+ CPU is used for this purpose
4. When maximizing the system idle state time as discussed in [Clocks](#), it may be beneficial to spread tasks between CM0+ and CM4 so that the lower-power idle state can be entered faster due to simultaneous task processing
5. If single CPU processing must be spread out over a long period due to real-time hardware interactions that limit time spent in lower-power states, CM0+ will often be lower-power than CM4 over the same active time interval
6. If processing power is required that exceeds what one CPU can provide, the tasks should be spread between both cores. Because each core has its own bus controller and can access separate memory blocks, there are minimal wait states due to bus or memory contention. The performance decrease due to contention is typically less than 2% while the performance increase from the second core is greater than 50% (assuming CM0+ added to CM4)
7. In applications that stay in active mode and can achieve the required processing power on a single core, there is most often no power benefit to sharing the required tasks between cores

~~DO NOT DRAFT~~

5 PSoC™ 6 application notes

5.9.4.7 Clocks

In some cases, running CPU clocks faster can result in a lower average current consumption. For example, consider a design that takes a reading from a sensor once every second, performs several calculations, and then transmits the results to another device.

You can use CPU sleep or system deep sleep mode to reduce power when the PSoC™ device is idle, but the average current consumption can be higher because of the additional time spent in CPU active mode. [Figure 395](#) is a representation of the current consumption of this example with the system clocks set at 8 MHz.

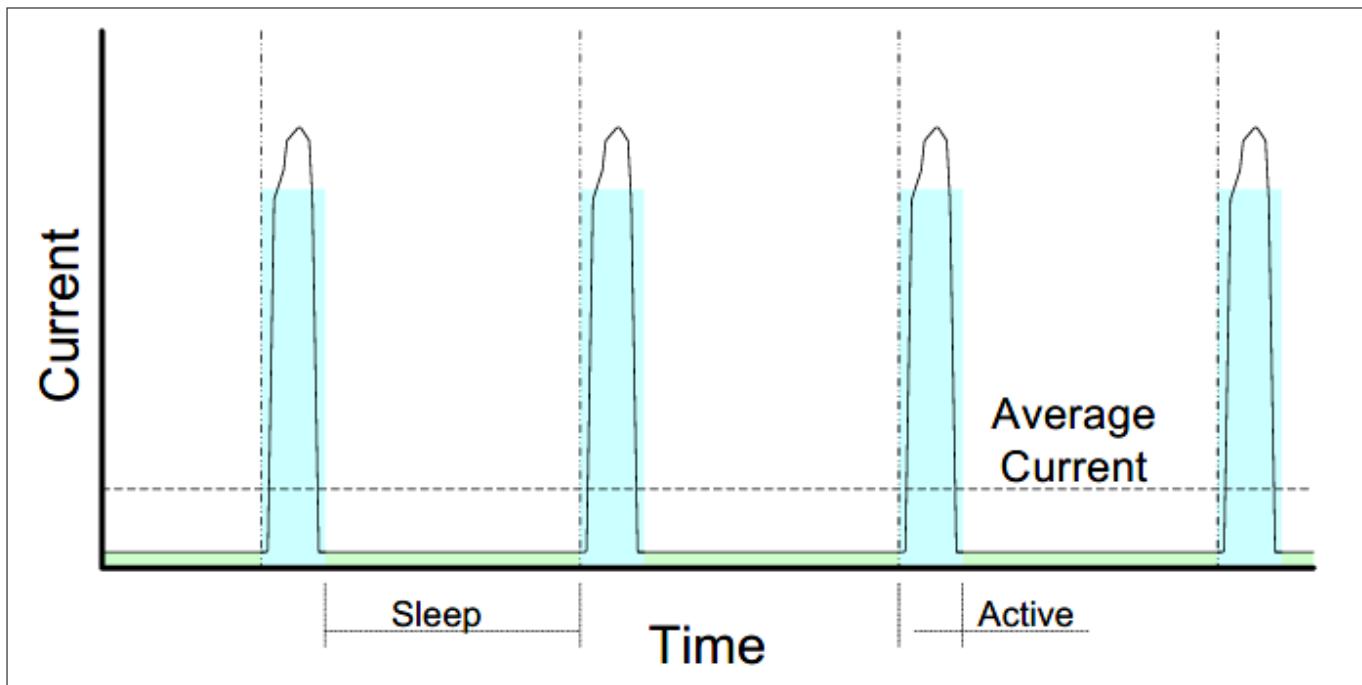


Figure 395 Example current profile with 8 MHz clocks

Depending on the tasks or calculations that are being performed when the PSoC™ device is awake, it may be possible to complete them sooner by running the CPU clocks faster. This can reduce the average current consumption because the PSoC™ device is in CPU active mode for less time and in CPU sleep or system deep sleep longer. [Figure 396](#) is a representation of CPU active mode timing, broken up into tasks.

5 PSoC™ 6 application notes

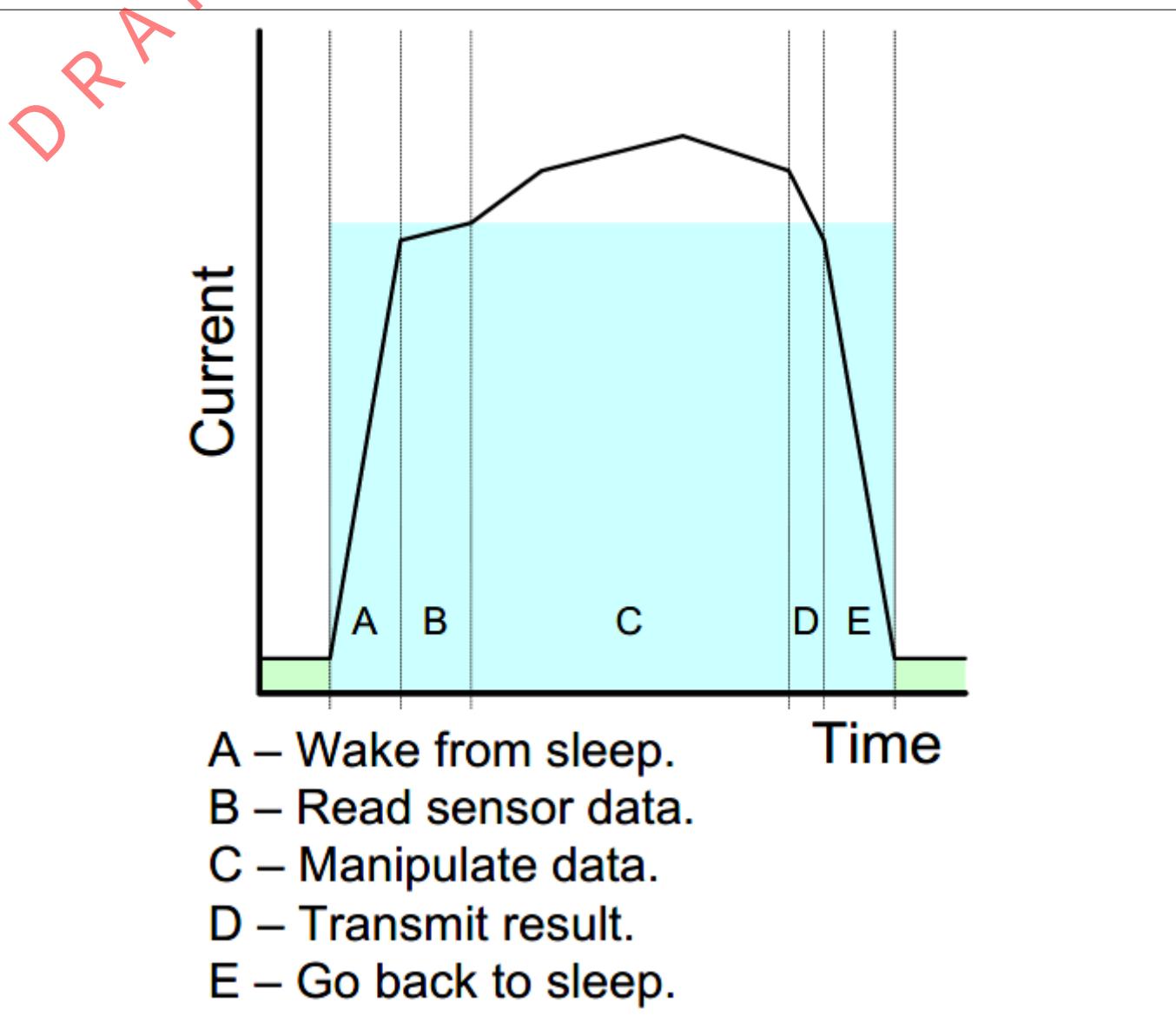


Figure 396 Analysis of tasks in CPU active mode at 8 MHz

The time required for some tasks does not change even if the system clock frequency increases. Sensor reading and data transmitting fall into this category. Other tasks like data processing, require less time if the CPU operates at a faster frequency.

At some point, the benefit of a shorter active time is overcome by the energy required to drive the clocks at a higher rate. Assume that the optimal speed is 48 MHz, as [Figure 397](#) shows. With a 48 MHz clock, the time spent in active mode is about half the time spent with an 8 MHz clock. [Figure 397](#) shows that the peak current consumption is greater when the clocks are faster, but the overall average is lower due to the longer time spent in CPU sleep or system deep sleep. On PSoC™ 6 MCU devices, it is generally better to run the CPU as fast as possible to complete processing and spend more time in the low-power mode.

5 PSoC™ 6 application notes

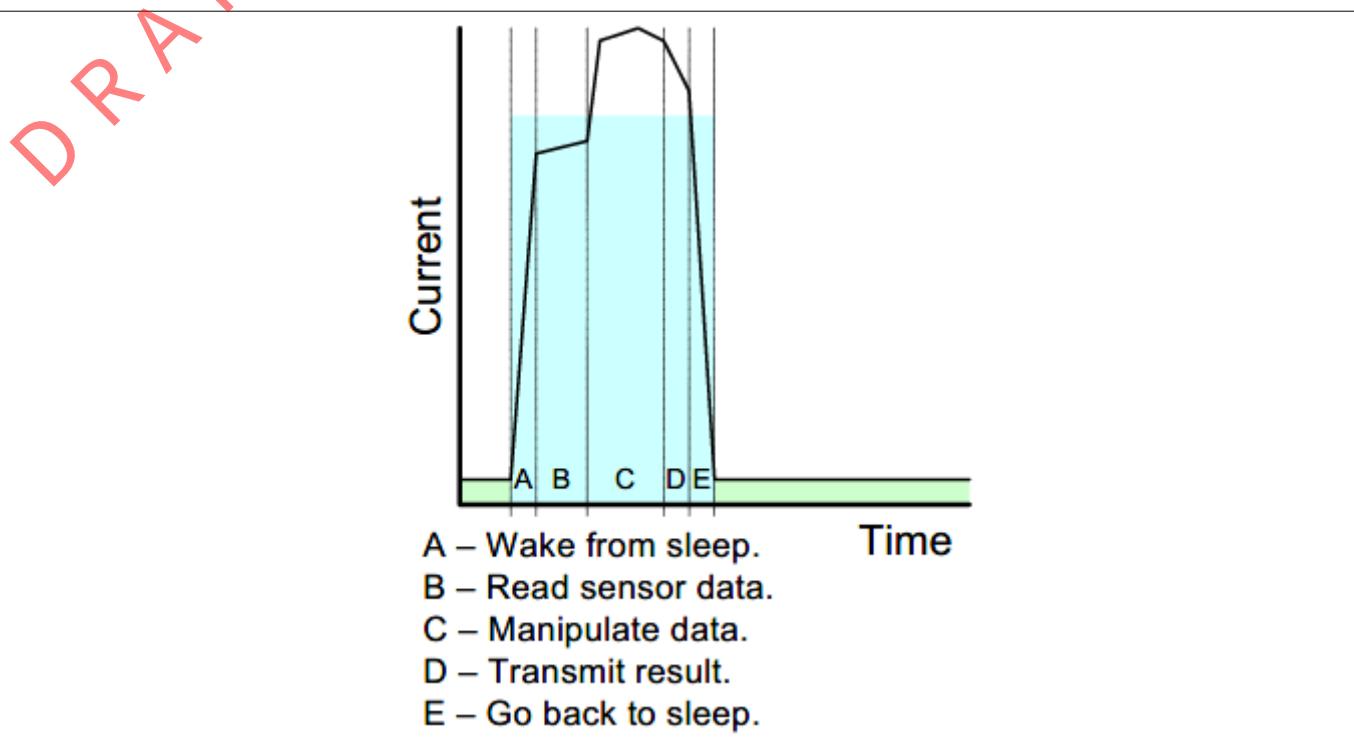


Figure 397 Analysis of tasks in active mode at 48 MHz

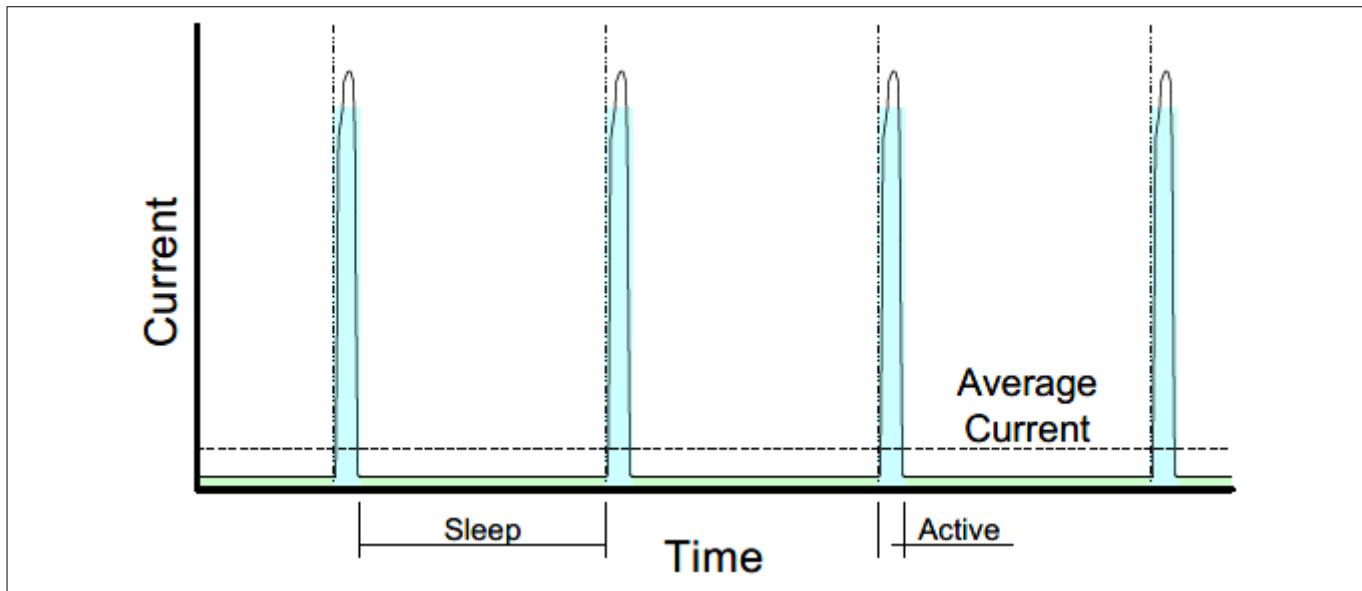


Figure 398 Example current profile with 48 MHz clocks

5.9.4.8 GPIOs

GPIOs can continue to drive external circuitry while the PSoC™ device is in any low-power mode. This is helpful when you need to hold external logic at a fixed level, but it can lead to wasted power if the pins needlessly source or sink current. The specific power savings of this technique depend on the circuit attached to the specific GPIO pin.

~~5 PSoC™ 6 application notes~~

You should analyze your design and determine the best state for your GPIOs during low-power operation. If holding a digital output pin at logic ‘1’ or ‘0’ results in the lowest current, match that digital level using Cy_GPIO_Write().

```
/* Set MyPin to '0' for low power. */
Cy_GPIO_Write(MYPIN_0_PORT, MYPIN_0_NUM, 0u);
```

Configure all unused GPIOs to Analog High-Z unless there is a specific reason to use a different drive mode. The High-Z drive mode results in the lowest current for a GPIO pin. A pin’s drive mode may be set using the Cy_GPIO_SetDrivemode() function.

```
/* Set MyPin to Alg HI-Z for low power. */
Cy_GPIO_SetDrivemode(MYPIN_0_PORT, MYPIN_0_NUM, CY_GPIO_DM_HIGHZ);
```

The flexibility of PSoC™ device makes it easy to manage GPIO drive modes to prevent unwanted current leakage. In system hibernate mode, the GPIO drive modes and data registers are automatically “frozen.” They must be reconfigured to a known state before being “unfrozen” to avoid GPIO output transitions after a wakeup reset and to allow their states to be changed at runtime. Call cy_SysPm_IoUnfreeze() after configuring the GPIO pins and setting their output drive state. See [Appendix D.2](#) for a code example on hibernate and I/O control.

5.9.4.9 SRAM

PSoC™ 6 MCU devices allow powering off individual SRAM banks or pages within a bank. The size of pages within a bank depends on the specific device and bank as detailed in the device datasheet. Specific devices will have one or more SRAM banks. Most devices have one bank with a smaller page size (typically 32 KB) for fine-grained control of the amount of SRAM enabled. Any unused page can be disabled by writing to the CPUSS power control registers. This technique is most useful in system deep sleep mode where retaining 64 K SRAM = 7 µA (SIDDS1) while retaining 256 K SRAM = 9 µA (SIDDS2) as listed in the CY8C61x6 datasheet.

```
/* Power off all except the first two pages of RAM0 */
for (uint32_t i = 2; i < NUM_RAM_PAGES; i++)
{
    CPUSS->RAM0_PWR_MACRO_CTL[i] = 0x05FA0000;
}
/* Additional SRAM banks */
CPUSS->RAM1_PWR_CTL = 0x05FA0000;
CPUSS->RAM2_PWR_CTL = 0x05FA0000;
```

Consult the register TRM of the device for all power control registers.

As an advanced use case, modification of the project’s linker script can be used with SRAM power down to optimize the total amount of SRAM required or allow custom data placement in support of dynamic SRAM powering.

5.9.4.10 TCPWM

When using a counter, timer or PWM, you should configure the clock sourcing the channel as low as possible while still meeting your frequency and accuracy requirements. For example, if you need to generate a 1-second interrupt with a timer, it is better to use a clock frequency of 1 kHz with the period equaling 1,000 counts than a clock frequency of 1 MHz with a period equal to 1,000,000 counts. The power savings from reducing the TCPWM

~~5 PSoC™ 6 application notes~~

clock is mostly linear based on clock frequency. TCPWM operating current at 100 MHz = 540 μ A (SID.TCPWM.2B) while at 8 MHz = 70 μ A (SID.TCPWM.1) per the CY8C61x6 datasheet.

The same idea applies when using a PWM to dim an LED. Use the minimum clock frequency possible that does not let the human eye perceive that the LED is blinking. 60 Hz is a good frequency for most applications. [Figure 399](#) shows a comparison between the clock settings for the TCPWM block.

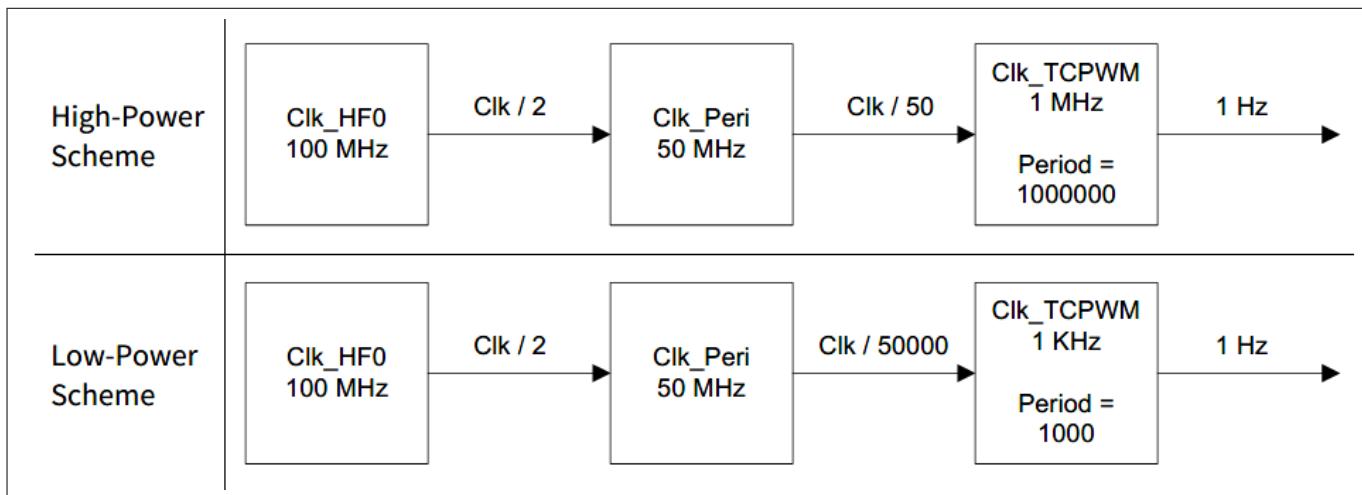


Figure 399 TCPWM clock settings comparison

The TCPWM block has a clock prescaler feature. For minimum power consumption, maximize the peripheral clock divider first before using the TCPWM clock prescaler.

If the TCPWM block uses any pin connections, set the pins to Analog High-Z when the block is disabled. If the PSoC™ device supports digital system interconnect (DSI), avoid using DSI by choosing the TCPWM channels with direct connection to the desired pin assignment. If the PSoC™ device supports UDBs (universal digital block), use all fixed-function TCPWM channels first instead of UDB-based TCPWMs.

5.9.4.11 SCB

Avoid using blocking functions when sending or receiving data. Use interrupt-based events or an RTOS to transfer the data while yielding the CPU to other tasks. The idea behind this strategy is to keep the CPU in a sleep state longer, instead of polling the status of the transmission. [Figure 400](#) shows an example of blocking and non-blocking functions.

5 PSoC™ 6 application notes

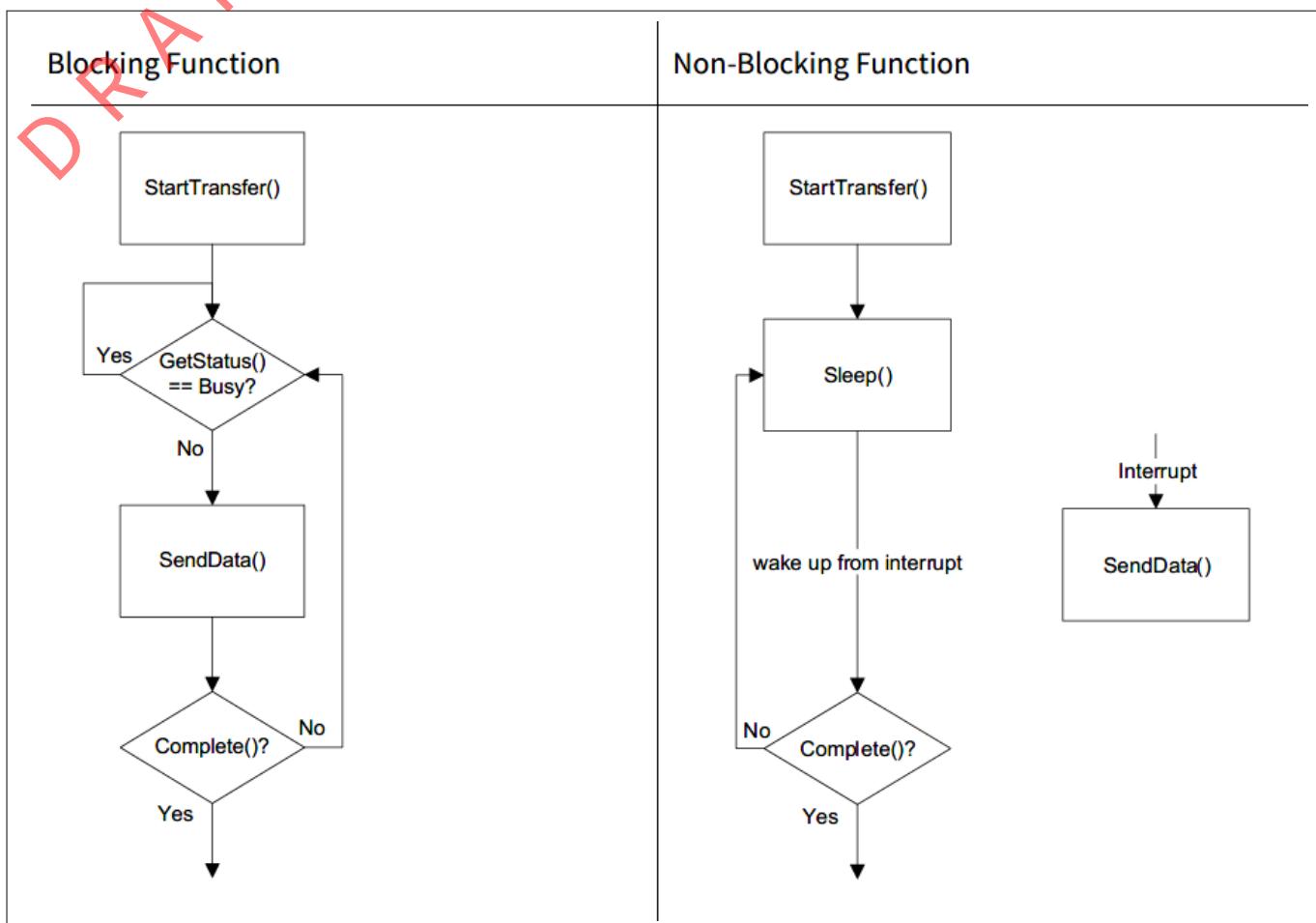


Figure 400 Example of blocking and non-blocking functions

Instead of using the SCB interrupt to access its FIFO in firmware, use DMAs controlled by the FIFO level to reduce the amount of CPU cycles required in the application; therefore, the CPU can stay in a sleep state longer or execute other tasks.

If the SCB block uses any pin connections, set the pins to Analog High-Z when the block is disabled. If the PSoC™ device supports DSI, avoid using it by choosing SCB channels with direct connections to the desired pin assignment. If the PSoC™ device supports UDBs, use all fixed-function SCB channels before using UDB-based communication interfaces.

Operating at the lowest data rate the system can support also helps in reducing power because the SCB's power increases linearly with clock frequency. For example, running I2C at 1 Mbps = 180 μ A (SID152) while running at 100 kbps = 30 μ A (SID149) as shown in the CY8C61x6 datasheet.

5.9.4.12 Audio subsystem

The I2S and PDM/PCM blocks are generally sourced by a high-frequency clock generated from the PLL. For high-accuracy frame rates, an ECO sources the PLL. When picking a frequency for the ECO, consider what the desired audio sample rates are:

- To support all standard audio sample rates (8/16/22.05/32/44.1/48 kHz), the minimum ECO frequency you can use is 17.2032 MHz
- To support only the sample rates 8/16/48 kHz, configure the PLL for 12.288 MHz
- To support only the sample rates 22.05/44.1 kHz, configure the PLL for 22.5792 MHz

If the Audio Subsystem is not in use disable the I2S and/or PDM/PCM blocks. You must also disable the PLL and ECO, which are the main reasons for high current consumption.

~~5 PSoC™ 6 application notes~~

If the accuracy of the frame rates is not important for the application, do not use the ECO and/or PLL. For example, if your application implements a sound detector and can tolerate a higher error, use the FLL as the source for the audio subsystem. Once a sound is detected and the system needs to be fully operational, enable the PLL and ECO to sample at a higher accuracy.

Instead of using audio subsystem interrupts to access its FIFO in firmware, use DMAs controlled by the FIFO level triggers to reduce the number of CPU cycles required in the application so that the CPU can stay in a sleep state longer or execute other tasks.

5.9.4.13 USB

When there is no activity on the USB bus, you can suspend the USB block to consume less current. See the code example [CE223305 – PSoC™ 6 MCU: USB suspend and resume](#) to learn how to set up the device to enter and exit low-power states.

5.9.4.14 Low-power comparator

If the low-power comparator requires pin connections and system deep sleep operation, always choose the dedicated pin connections to the comparator. This allows the comparator to work in system deep sleep and avoid using the global analog muxes, which consume additional current to stay active.

There are two ways to verify the status of the comparator: Call the `Cy_LPCOMP_GetCompare()` function, or set up an interrupt. In most cases, it is preferred to use the interrupt method, which reduces the CPU usage.

The comparator supports three operation modes that affect its speed and power:

1. **Ultra-low-power/Slow:** $I_{CMP1} = 0.85 \mu A$ (SID259) – Slows the response time to 20 μs (SID92). Use this mode when in system deep sleep or hibernate only
2. **Low-power/Low:** $I_{CMP2} = 10 \mu A$ (SID248) – Preferred mode when system deep sleep or hibernate are not required and the slower response time of 1 μs (SID258) is acceptable. This increases the input offset voltage to 25 mV (SID85A)
3. **Normal Power/Fast:** $I_{CMP1} = 150 \mu A$ (SID89) – Use it when the fastest response time of 100 ns (SID91) and/or a low offset voltage of 10 mV (SID84) are required

Another option to reduce the amount of current consumed by the comparator is to disable hysteresis.

Hysteresis can be disabled in the ModusToolbox™ software configurator, PSoC™ Creator IDE customizer, or PDL configuration structure.

5.9.4.15 SAR ADC

When selecting the pins to be connected to the SAR ADC, preferably choose the dedicated port connected to the SAR MUX first. Only after running out of pins in that port, choose pins from other ports. By only using the dedicated port, there is no need to use the global analog muxes, which consume extra current. You can also achieve higher sample rates with the dedicated port due to lower input capacitance.

If the full-rated accuracy of ADC results is not required, use a lower resolution and do not use averaging, which reduces the number of ADC clocks required for the same sample rate.

If the maximum sample rate is not required, consider using the single-shot mode instead of continuous mode. This avoids the SAR ADC operating all the time. In single shot mode, The ADC samples only when triggered by software or hardware, depending on the application.

If the main application is not required to perform processing during ADC sampling, the CY8C62x4 family of devices support SAR ADC operation in system deep sleep power mode. See [AN230938 – PSoC™ 6 MCU low-power analog](#) for details of the low-power analog features of the CY8C62x4 family.

~~5 PSoC™ 6 application notes~~

~~5.9.4.16~~ Voltage DAC

If the DAC voltage output needs to be periodically changed with pre-determined values, use a DMA to update the voltage value. This avoids using CPU cycles to write to DAC registers.

If the external device that requires the DAC output voltage is used only periodically, keep the DAC disabled when its output is not required.

When the DAC output voltage must be provided while in system deep sleep, use a “sample-and-hold” strategy to maintain the voltage output while the voltage DAC is disabled in system deep sleep. See the code example [CE220925 – PSoC™ 6 MCU VDAC sample and hold](#) for more details. The DAC operating current = 125 µA (SID100D) while the DAC current stopped = 1 µA (SID101D) per the CY8C61x6 datasheet.

5.9.4.17 Opamp

The opamp supports three different operating modes – Low, Medium, and High. These operation modes are affected by the system power modes – system LP/ULP and deep sleep. In system deep sleep, there are two additional modes to choose from. [Table 81](#) shows how to choose the appropriate operation mode.

Table 81 Opamp operational mode conditions

Specification	High-power operation mode		Medium-power operation mode		Low-power operation mode	
	LP/ULP	Deep sleep	LP/ULP	Deep sleep	LP/ULP	Deep sleep
Minimum gain-bandwidth	6 MHz	M1: 4 MHz M2: 0.5 MHz	4 MHz	M1: 2 MHz M2: 0.2 MHz	1 MHz	M1: 0.5 MHz M2: 0.1 MHz
Maximum output current (no load)	1500 µA	M1: 1500 µA M2: 120 µA	600 µA	M1: 600 µA M2: 60 µA	350 µA	M1: 350 µA M2: 15 µA
Response time	150 ns		400 ns		2000 ns	

M1: Mode 1 has higher GBW and highest current.

M2: Mode 2 has a lower GBW and the lowest current. When selecting the pins to be connected to the opamp, preferably choose the dedicated pins connected to the opium. By using only the dedicated pins, there is no need to use the global analog muxes, which consume extra current.

5 PSoC™ 6 application notes**5.9.5 Power measurement****5.9.5.1 Measuring the current with a DMM**

~~DRAFT~~

When using a digital multimeter (DMM) to measure device current, it is important to know the value of the shunt resistor in the DMM. DMMs have one or more (shunt) resistors between the current inputs. These resistors can range from less than an ohm to more than 10 kΩ. There is no standard value for the shunt resistor between brands or even models from the same vendor. It is important to review your meter's manual and learn the value of the shunt resistor because there will always be a voltage drop across this shunt. This means that the PSoC™ device will not see the same voltage you think you are supplying. If the shunt resistor in your meter is 1 Ω or less, you will see only a few millivolts of drop when measuring PSoC™ MCU current, which can be ignored. If the shunt resistor is 1 kΩ, which some vendors use for low-current measurements, a 1 mA current will result in a drop of 1 V.

High shunt resistor values can cause the device to reset due to low voltage at higher currents. Also, when changing ranges, be careful that the DMM does not do a “break-before-make”, or the power will be cycled and your project will be reset.

For extremely low currents in deep sleep, hibernate, and stop modes, a good technique is to use a zero or low-resistance shunt until the device enters low-power mode. After entering low-power mode, the code should keep the device in that mode and switch to a high-resistance shunt for current measurement.

As an alternative to relying on the DMM shunt, most PSoC™ 6 MCU kits include a place for a shunt resistor or current measurement header. See the respective kits user guide to determine specific current measurement features. These can be replaced or connected to a small resistor allowing measurement of the voltage across the shunt with a voltmeter. You can then easily determine the current. A shunt between 1 Ω and 100 Ω should work well for most applications.

5.9.5.2 Approximating the power consumption

The device datasheet provides information to estimate PSoC™ 6 MCU power consumption for project-specific user configurations. To simplify this process, a spreadsheet has been provided that includes typical power requirements for a wide range of internal components and modes. The spreadsheet includes CY8C62x4 devices which contain low-power analog peripherals detailed in *AN230938 – PSoC™ 6 MCU low-power analog*. This spreadsheet, *PSoC_6_Power_Estimator.xlsx*, is located on the [AN219528 page](#). Because every project is different, the power calculation provided by this spreadsheet is only an estimate of typical values but should be close enough to provide feedback before your design is complete.

There are several tabs in the spreadsheet; make sure that you read the “Instructions” tab before entering your data. Five “Config” tabs allow configuration of different hardware settings and power modes that the device can transition through at run time. The “Summary” tab displays the current for each configuration as well as the peak and average current across all modes based on the time spent in each configuration. An optional battery life estimation section on the “Summary” tab estimates battery life based on the average current.

5 PSoC™ 6 application notes**5.9.6 Power supply protection system****5.9.6.1 Hardware control****5.9.6.1.1 Brownout detect (BOD)**

Brownout detect (BOD) can reset the system before the logic state is lost when V_{DDD} and V_{CCD} power are lost. The brownout system guarantees a reset before V_{DDD} reaches the minimum system operating voltage, and works for all logic, SRAM, and flash. BOD is controlled by hardware; there is no configurable register.

5.9.6.1.2 Low-voltage detect (LVD)

Low-voltage detect (LVD) is similar to BOD but its threshold voltage is configurable. LVD generates an interrupt when V_{DDD} falls under a configured trip voltage. This interrupt allows the application time to handle important data before the BOD reset is triggered. LVD provides 15 selectable trip voltages. LVD is not available in system deep sleep and hibernate modes.

5.9.6.1.3 Overvoltage detect (OVD)

Overvoltage detect (OVD) is the opposite of BOD. These circuits generate a reset when unsafe overvoltage supply conditions on V_{DDD} and V_{CCD} are detected. No firmware control is required. OVD helps to protect the system from high-voltage damage.

5 PSoC™ 6 application notes**5.9.7 Summary**

Many power managing options can be used in PSoC™ 6 MCU. By following proper methods, you can optimize your design and ensure that the power modes and features of PSoC™ 6 MCU give the best options for the lowest power consumption without degrading the required performance of battery powered devices.

5 PSoC™ 6 application notes
A Power modes summary
A.1 Power modes and wakeup source
Table 82 Power modes and wakeup source

System power mode	MCU power mode	Description	Entry conditions	Wakeup sources	Wakeup action
LP	Active	Primary mode of operation. 1.1 V core voltage. All peripherals are available (programmable). Clocks at their maximum frequencies.	<ul style="list-style-type: none"> Reset from external reset, brownout, power on reset system and hibernate mode Manual register write from system ULP mode Wakeup from CPU sleep or CPU deep sleep while in system LP mode Wakeup from system deep sleep after entered from LP mode 	Not applicable	N/A
	Sleep	1.1 V core voltage. One or more CPUs in sleep mode (execution halted). All peripherals are available (programmable). Clocks at their maximum frequencies.	In system LP mode, the CPU executes the WFI/WFE instruction with deep sleep disabled	Any interrupt to CPU	Interrupt
	Deep sleep	1.1 V core voltage. One CPU in deep sleep mode (execution halted). Other CPU in active or sleep mode. All peripherals are available (programmable). Clocks at their maximum frequencies.	In system LP mode, the CPU executes the WFI/WFE instruction with deep sleep enabled	Any interrupt to CPU	Interrupt
ULP	Active	0.9 V core voltage. All peripherals are available (programmable). Clock frequencies are limited.	Manual register write from system LP mode. Wakeup from CPU sleep or CPU deep sleep while in system ULP mode. Wakeup from system deep sleep after entered from ULP mode.	Not applicable	N/A

(table continues...)

DRAFT

5 PSoC™ 6 application notes

Table 82 (continued) Power modes and wakeup source

System power mode	MCU power mode	Description	Entry conditions	Wakeup sources	Wakeup action
	Sleep	0.9 V core voltage. One or more CPUs in sleep mode (execution halted). All peripherals are available (programmable). Clock frequencies are limited.	In system ULP mode, the CPU executes the WFI/WFE instruction with deep sleep disabled	Any interrupt to CPU	Interrupt
	Deep sleep	0.9 V core voltage. One CPU in deep sleep mode (execution halted). Other CPU in active or sleep mode. All peripherals are available (programmable). Clock frequencies are limited.	In system ULP mode, the CPU executes the WFI/WFE instruction with deep sleep enabled	Any interrupt to CPU	Interrupt
Deep sleep	Deep sleep	All high-frequency clocks and peripherals are turned off. Low-frequency clock (32 kHz) and low-power analog and digital peripherals are available for operation and as wakeup sources. SRAM is retained (programmable).	Both CPUs simultaneously in CPU deep sleep mode	GPIO interrupt, low-power comparator, SCB, CTBm, watchdog timer, and RTC alarms	Interrupt
Hibernate	N/A	GPIO states are frozen. All peripherals and clocks in the device are completely turned OFF except the low-power comparator and backup domain. Wakeup is possible through WAKEUP pins, XRES, low-power comparator (programmable), and RTC alarms (programmable). Device resets on wakeup.	Manual register write from LP or ULP modes	WAKEUP pin, low-power comparator, watchdog timer ²⁸ , and RTC ²⁹ alarms.	Reset

²⁸ Watchdog timer is capable of generating a hibernate wakeup.

²⁹ RTC (along with WCO) is a part of the backup domain and is available irrespective of the device power mode. RTC alarms are capable of waking up the device from any power mode

5 PSoC™ 6 application notes
B Subsystem availability
B.1 Resources available in different power modes

Table 83 shows information for the resource availability in different power modes.

Table 83 Resources available in different power modes

Component	System power modes							
	LP		ULP		Deep sleep	Hibernate	XRES	Power off with backup
	CPU active	CPU sleep/ deep sleep	CPU active	CPU sleep/ deep sleep				
Core functions								

CPU	On	Sleep	On	Sleep	Retention	Off	Off	Off
SRAM	On	On	On	On	Retention	Off	Off	Off
Flash	Read/Write	Read/Write	Read only	Read only	Off	Off	Off	Off
High-speed clock (IMO, ECO, PLL, FLL)	On	On	On	On	Off	Off	Off	Off
LVD	On	On	On	On	Off	Off	Off	Off
ILO	On	On	On	On	On	On	Off	Off

Peripherals

SMIF	On	On	On	On	Retention	Off	Off	Off
UDB	On	On	On	On	Off	Off	Off	Off
SAR ADC	On	On	On	On	Off	Off	Off	Off
CTBm	On	On	On	On	On (lower GBW) ³⁰⁾	Off	Off	Off
LPCMP	On	On	On	On	On ³⁰⁾	On ³¹⁾	Off	Off
TCPWM	On	On	On	On	Off	Off	Off	Off
CSD	On	On	On	On	Retention	Off	Off	Off
Bluetooth® LE	On	On	On	On	Retention	Off	Off	Off
LCD	On	On	On	On	On	Off	Off	Off

(table continues...)

³⁰ Low-power comparator and CTBm may be optionally enabled in system deep sleep mode to generate wakeup.

³¹ Low-power comparator may be optionally enabled in hibernate mode to generate wakeup.

5 PSoC™ 6 application notes

Table 83 (continued) Resources available in different power modes

Component	System power modes							
	LP		ULP		Deep sleep	Hibernate	XRES	Power off with backup
	CPU active	CPU sleep/ deep sleep	CPU active	CPU sleep/ deep sleep				
SCB	On	On	On	On	Retention (I2C/SPI wakeup available) ³²⁾	Off	Off	Off
GPIO	On	On	On	On	On	Freeze	Off	Off
Watchdog timer	On	On	On	On	On	On	Off	Off
Multi-counter WDT	On	On	On	On	On	Off	Off	Off
Resets								
XRES	On	On	On	On	On	On	On	Off
POR	On	On	On	On	On	On	Off	Off
BOD	On	On	On	On	On	Off	Off	Off
Watchdog reset	On	On	On	On	On	On ³³⁾	Off	Off
Backup domain								
WCO, RTC, alarms	On	On	On	On	On	On	On	On

³² Only one SCB with deep sleep support is available in system deep sleep power mode; other SCBs are not available in system deep sleep power mode.

³³ Watchdog interrupt can generate a hibernate wakeup. See the “Watchdog Timer” chapter of the TRM for details.

5 PSoC™ 6 application notes**C Callback function examples****C.1 Register callback functions**

DRAFT

```
cy_stc_syspm_callback_params_t myParams;
cy_stc_syspm_callback_t myAppSleep =
{
    &Application_Callback,          /* Callback function */
    CY_SYSPM_SLEEP,                /* Select Power Mode */
    CY_SYSPM_SKIP_CHECK_READY |   /* Skip CHECK_READY and CHECK FAIL */
    CY_SYSPM_SKIP_CHECK_FAIL),     /* Operation, contexts */
    &myParams,                     /* Previous list callback */
    NULL,                          /* Next list callback */
};

cy_stc_syspm_callback_t myAppHibernate =
{
    &Application_Callback,          /* Callback function */
    CY_SYSPM_HIBERNATE,             /* Select Power Mode */
    0U,                            /* Skip mode, no skip */
    &myParams,                     /* Operation, contexts */
    NULL,                          /* Previous list callback */
    NULL                           /* Next list callback */
};

/* Register Callback functions for each power mode */
Cy_SysPm_RegisterCallback(&myAppSleep);
Cy_SysPm_RegisterCallback(&myAppHibernate);
```

5 PSoC™ 6 application notes**C.2 Implement custom callback functions**

DRAFT

```
cy_en_syspm_status_t Application_Callback(
    cy_stc_syspm_callback_params_t *callbackParams)
{
    cy_en_syspm_status_t retVal = CY_SYSPM_SUCCESS;
    switch(callbackParams->mode)
    {
        case CY_SYSPM_CHECK_READY:
        {
            if(Check_HW())
            {
                /* Hardware is ready */
            }
            else
            {
                retVal = CY_SYSPM_FAIL;
            }
        }
        break;
        case CY_SYSPM_CHECK_FAIL:
        {
            /* Rollback any configuration during CHECK_READY */
            Rollback_HW();
            retVal = CY_SYSPM_SUCCESS;
        }
        break;
        case CY_SYSPM_BEFORE_TRANSITION:
        {
            /* configure HW for new mode before transition */
            ConfigureHW_BeforeMode();
            retVal = CY_SYSPM_SUCCESS;
        }
        break;
        case CY_SYSPM_AFTER_TRANSITION:
        {
            /* configure HW after mode transition */
            ConfigureHW_AfterMode();
            retVal = CY_SYSPM_SUCCESS;
        }
        break;
        default:
        break;
    }
    return (retVal);
}
```

5 PSoC™ 6 application notes

D Code examples

D.1 CE219881 - PSoC™ 6 MCU switching between power modes

This code example shows how to enter and exit system LP and ULP power modes and transition the CPU from CPU active to sleep or deep sleep. Once in either mode, the example also shows how to wake up and return to one of the system LP or ULP modes and CPU active mode.

The project uses a switch to transition among power modes and shows different LED colors to indicate the current power mode. [Figure 401](#) shows the state machine implemented in the firmware to execute the transitions. For more information, see [CE219881 - PSoC™ 6 MCU switching between power modes](#).

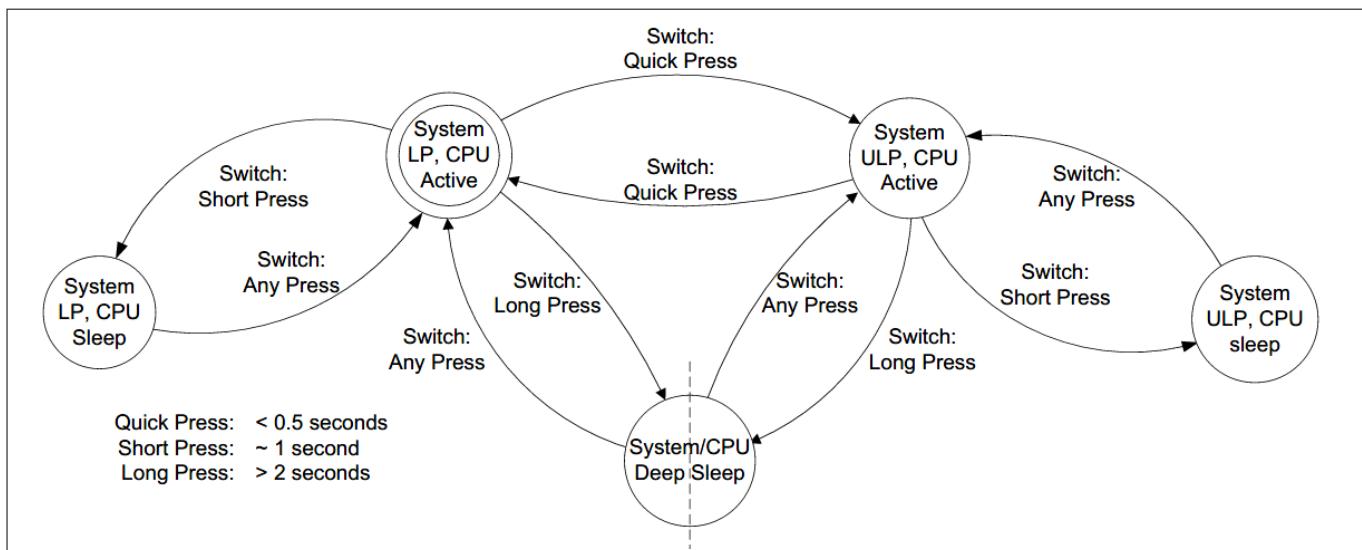


Figure 401 Power mode state machine

D.2 CE218129 - PSoC™ 6 MCU wakeup from hibernate using a low-power comparator

This code example demonstrates how to set the Component options for the LPComp internal reference voltage and how to set the external input from a GPIO using the LPComp driver. The project is a good example of the system hibernate power mode transition. It teaches you how to handle the GPIOs before and after system hibernate, and it shows how to register the wakeup source for hibernate. [Figure 402](#) shows the basic flow of the project. For more information, see [CE218129 - PSoC™ 6 MCU wakeup from hibernate using a low-power comparator](#).

5 PSoC™ 6 application notes

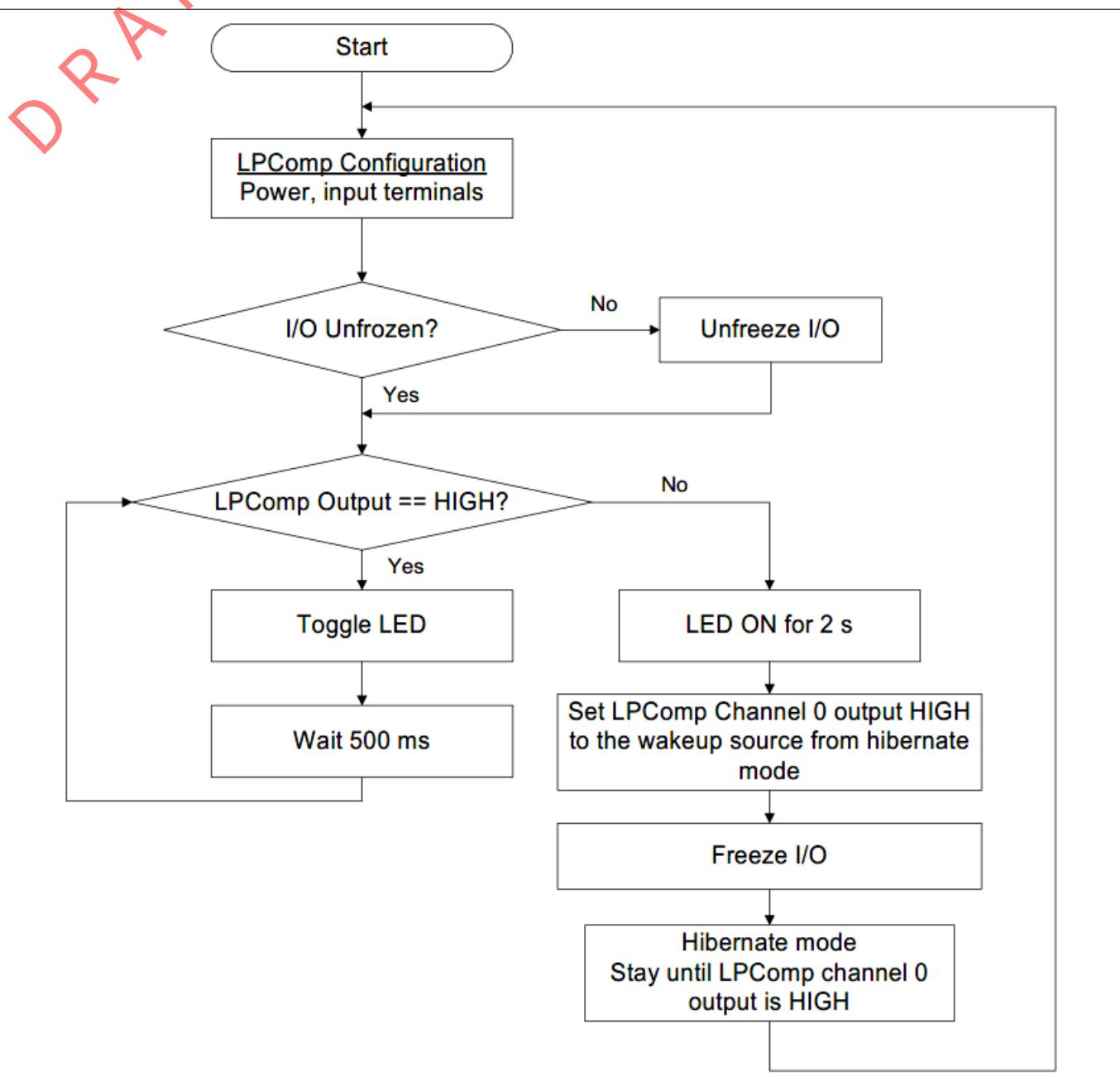


Figure 402

Wakeup from system hibernate mode using LPComp input

D.3 CE218542 - PSoC™ 6 MCU custom tick timer using RTC alarm interrupt

This code example demonstrates how to configure the RTC registers for a periodic alarm interrupt using the PDL RTC driver API. The project uses system LP and system deep sleep modes to reduce power consumption. A GPIO output is included to toggle the LED to show the period of the interrupt. For more information, see [CE218542 - PSoC™ 6 MCU custom tick timer using RTC alarm interrupt](#).

5 PSoC™ 6 application notes

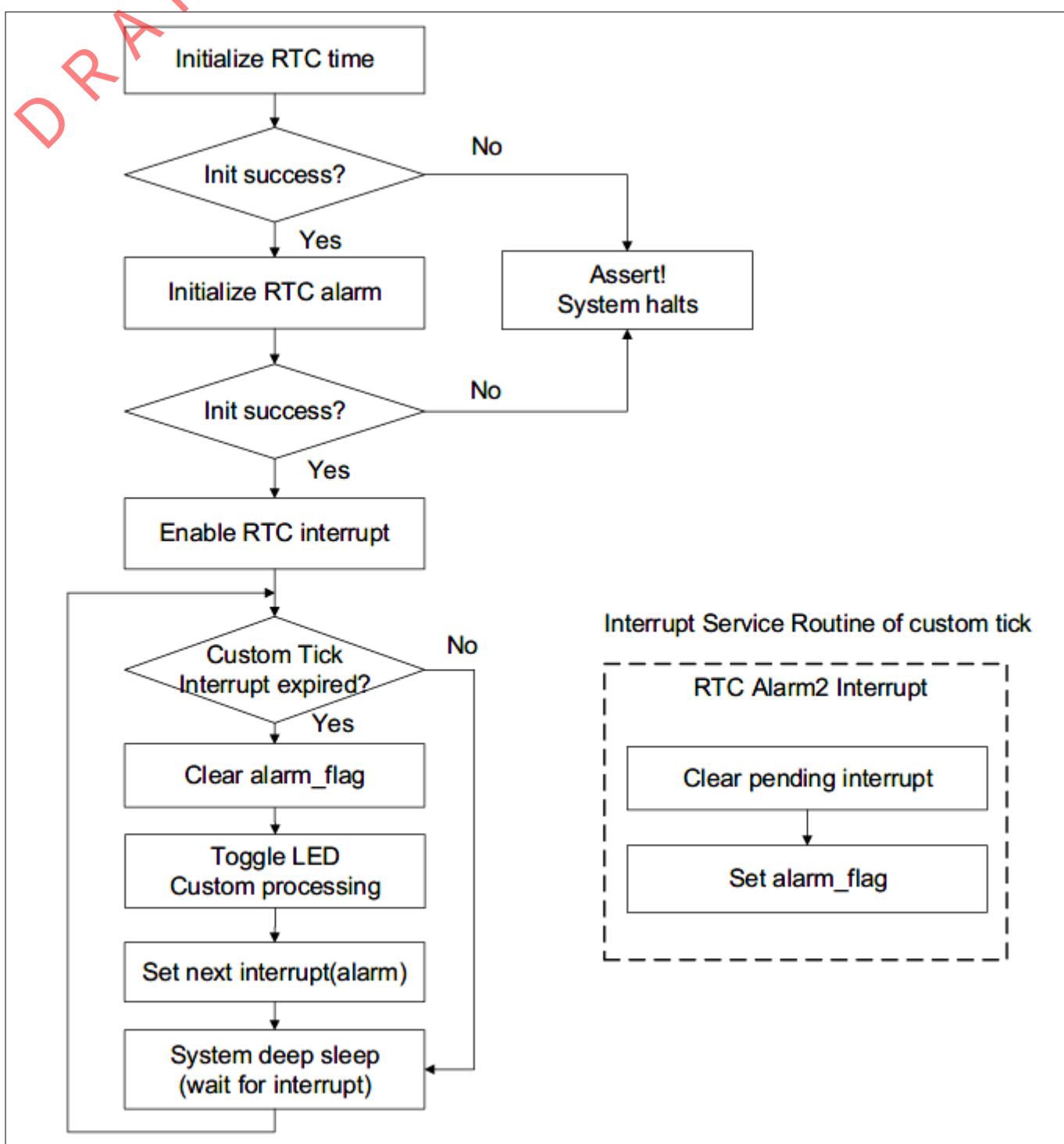


Figure 403 RTC periodic wakeup timer using alarm interrupt

D.4 CE226306 - PSoC™ 6 MCU power measurements

This example shows how to configure PSoC™ 6 MCU devices to run the clock frequencies and system modes specified in the device-level specifications table in PSoC™ 6 MCU datasheets. After building and programming the application into PSoC™ 6 MCU devices, you can measure the current the PSoC™ 6 MCU device consumes and compare it against the values specified in the datasheet. You can select the power configuration by changing a #define in firmware.

5 PSoC™ 6 application notes

~~DRAFT~~ **Table 84 Configuration options**

Configuration	Options	Description
System mode	SYSTEM_LP SYSTEM_ULP	Defines the system mode the firmware enters – system low-power or system ultra-low-power modes. The following functions are used: <div style="border: 1px solid black; padding: 5px;"> <code>Cy_SysPm_SystemEnterLp(); // Enter System Low Power mode Cy_SysPm_SystemEnterUlp(); // Enter System Ultra-Low Power mode</code> </div>
Core voltage supply	VCCD_1V1 VCCD_0V9	Defines the core voltage supply for the BUCK or LDO – 0.9 V or 1.1 V. <div style="border: 1px solid black; padding: 5px;"> <code>Cy_SysPm_SwitchToSimoBuck(); Cy_SysPm_SimoBuckSetVoltage1(CY_SYSPM_SIMO_BUCK_OUT1_VOLTAGE_1_1V);</code> </div>
Cache support	RUN_FROM_FLASH RUN_FROM_CACHE	Defines if cache is used. If cache is disabled, the following instructions are used: <div style="border: 1px solid black; padding: 5px;"> <code>//Disable CM4 cache //Disable CM0+ cache CY_SET_REG32(CYREG_FLASHC_CM0_CA_CTL0, CY_GET_REG32(CYREG_FLASHC_CM0_CA_CTL0) & CACHE_DISABLE_MASK);</code> </div> <p>The <code>cy_SysLib_SetWaitStates()</code> function is called based on the cache support. If disabled, set the maximum allowed frequency for the given System Mode. If cache is enabled, set based on the CM4 CPU frequency.</p>
CM4 CPU mode	CM4_WHILE_LOOP CM4_DHRYSTONE CM4_SLEEP CM4_DEEP_SLEEP	Defines what the CM4 CPU runs – While (1) loop, Dhrystone algorithm, go to CPU sleep, or go to CPU deep sleep. The following functions are used: <div style="border: 1px solid black; padding: 5px;"> <code>while (1); // Runs a forever while loop dhystone(); // Runs the Dhrystone instructions Cy_SysPm_CpuEnterSleep(CY_SYSPM_WAIT_FOR_INTERRUPT); // Sleep Cy_SysPm_CpuEnterDeepSleep(CY_SYSPM_WAIT_FOR_INTERRUPT); // Deep-Sleep</code> </div>
CM4 CPU frequency [CM4_FREQ_MHZ]	FREQ_8_MHZ FREQ_25_MHZ FREQ_50_MHZ FREQ_100_MHZ	Defines the clock frequency for HFCLK0, which is linked to the CM4 CPU clock. The following functions are used: <div style="border: 1px solid black; padding: 5px;"> <code>Cy_SysClk_FllConfigure(FREQ_8_MHZ, CM4_FREQ_MHZ, CY_SYSCLK_FLLPLL_OUTPUT_AUTO); // Configure the FLL Cy_SysClk_FllEnable(TIMEOUT_LOCK); // Enable the FLL</code> </div>

(table continues...)

5 PSoC™ 6 application notes

~~DRAFT~~ Table 84 (continued) Configuration options

Configuration	Options	Description
CM0+ CPU mode	CM0P_WHILE_LOOP CM0P_DHRYSTONE CM0P_SLEEP CM0P_DEEP_SLEEP CM0P_HIBERNATE	Defines what the CM0+ CPU runs – While (1) loop, Dhystone algorithm, go to CPU sleep, go to CPU deep sleep or hibernate The same functions from the CM4 CPU Mode are used. <code>Cy_SysPm_SystemEnterHibernate(); // Go to Hibernate</code>
CM0+ CPU frequency [CM0P_FREQ_MHZ]	FREQ_8_MHZ FREQ_25_MHZ FREQ_50_MHZ FREQ_100_MHZ	Defines the clock frequency for the CM0+ CPU clock. The following function is used: <div style="border: 1px solid black; padding: 5px;"> <code>/* Set the PERI Clock Divider */ Cy_SysClk_ClkPeriSetDivider((CM4_FREQ_MHZ/CM0P_FREQ_MHZ)-1);</code> </div>
Clock source	USE_IMO USE_FLL	Defines the clock source for the HCLK0 – IMO or FLL. If IMO is used, the FLL is bypassed. If the FLL is used, the IMO is still used as the source for the FLL. The following functions are used: <div style="border: 1px solid black; padding: 5px;"> <code>Cy_SysClk_ClkPathSetSource(0UL,CY_SYSCLK_CLKPATH_IN_IMO); Cy_SysClk_ClkHFSetSource(0UL,CY_SYSCLK_CLKHF_IN_CLKPATH0);</code> </div>
Minimum regulator current mode	MIN_CURRENT	Determines whether the firmware calls the <code>Cy_SysPm_SystemSetMinRegulatorCurrent()</code> function.

5 PSoC™ 6 application notes

5.9.12 References

Application notes

- [1] [AN215656 – PSoC™ 6 MCU dual-CPU system design](#)
- [2] [AN227910 – PSoC™ 6 MCU low- power system design with AIROC™ CYW43012 Wi-Fi & Bluetooth® combo chip and PSoC™ 6 MCU](#)

Code examples

- [1] [CE219881 – PSoC™ 6 MCU switching between power modes](#)
- [2] [CE226306 – PSoC™ 6 MCU power measurements](#)
- [3] [CE218542 – PSoC™ 6 MCU custom tick timer using RTC alarm interrupt](#)
- [4] [CE218129 – PSoC™ 6 MCU wakeup from hibernate using a low- power comparator](#)

5 PSoC™ 6 application notes

5.9.13 Revision history

Document version	Date of release	Description of changes
**	2017-09-12	New application note.
*A	2018-05-05	Updated power mode names to match TRM names; CPU active, CPU, sleep, CPU deep sleep, system LP, system ULP, system deep sleep, system hibernate.
*B	2019-12-19	<ul style="list-style-type: none"> Added CE226306 PSoC™ 6 MCU Power Measurements Added reference to AN227910 Low-Power System Design AIROC™ CYW43012 Wi-Fi & Bluetooth® combo chip and PSoC™ 6 MCU Added sections on Low Power Assistant, Power Estimator, measuring current with a DMM, and approximating power consumption Added design details for the following IP blocks: Disabling CPUs, Splitting tasks between the CPUs, SRAM, TCPWM, SCB Audio subsystem, USB, Low-power comparator, SAR ADC, Voltage DAC, Opamp. Added PSoC™ 6 Power Calculator Excel spreadsheet file download
*C	2020-11-23	Complete rewrite of Chapter 3.3 Backup Power Domain to add more details on its operation.
*D	2021-07-29	Added reference to AN230938 – PSoC™ 6 MCU low-power analog and added its content to the PSoC_6_Power_Estimator.xlsx power estimation spreadsheet.
*E	2022-07-21	Template update
*F	2023-01-23	No content change; added the "PSoC 6 Power Calculator (version 2).xlsx" file in the source of DMS.

5.10 AN215656 PSoC™ 6 MCU dual-core system design

About this document

•
1
0

Scope and purpose

AN215656 describes the dual-core architecture in PSoC™ 6 MCUs, which includes Arm® Cortex®-M4 and Cortex®-M0+ cores, as well as an inter-processor communication (IPC) module. A dual-core architecture provides the flexibility to help improve system performance and efficiency and reduce power consumption. The application note also shows how to build a simple dual-core design using ModusToolbox™ software, Command-line Interface (CLI), and PSoC™ Creator IDE, and how to debug the design using various IDEs.

To access an ever-growing list of hundreds of PSoC™ code examples, please visit our [code examples web page](#). You can also explore the Infineon video training library [here](#).

Intended audience

This application note is intended for advanced engineers who want to use the full dual-core capabilities of the PSoC™ 6 MCU.

5 PSoC™ 6 application notes~~DRAFT~~
5.10.1 Introduction

PSoC™ 6 MCU is 32-bit ultra-low-power PSoC™, purpose-built for the Internet of Things (IoT). It integrates low-power flash and SRAM technology, programmable digital logic, programmable analog, high-performance analog-digital conversion, low-power comparators, and standard communication and timing peripherals.

Of particular interest in PSoC™ 6 MCU is the CPU subsystem. The architecture incorporates multiple bus masters – two CPUs, DMA Controllers, and a cryptography block (Crypto) – as [Figure 404](#) shows:

5 PSoC™ 6 application notes

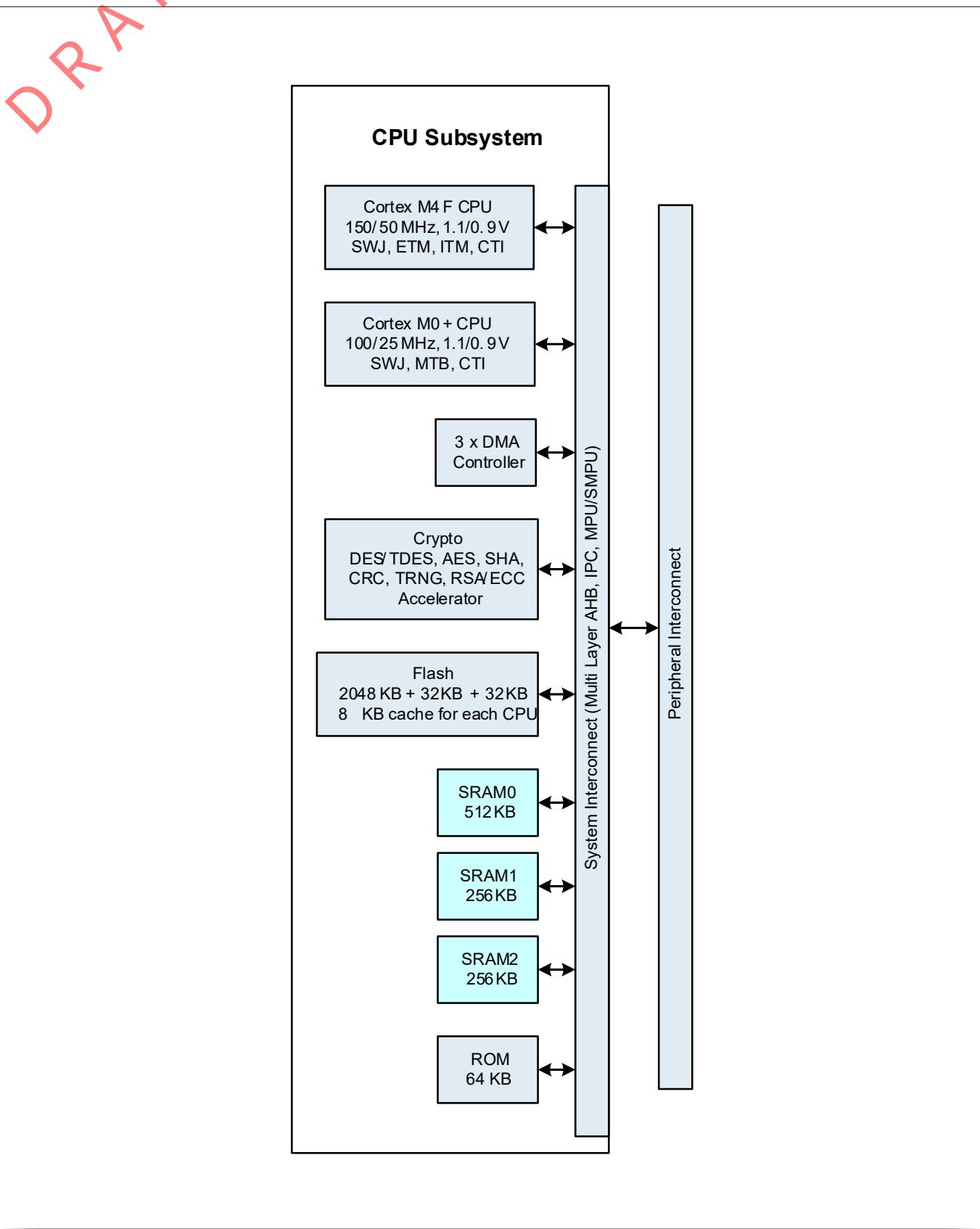


Figure 404 PSoC™ 6 MCU typical CPU subsystem architecture

5 PSoC™ 6 application notes

~~DRAFT~~

Note: The contents of the block diagram in [Figure 404](#) may vary depending on the device. Some PSoC™ 6 MCU parts have only one CPU available for application use. See the [device datasheet](#) for details. This application note does not apply to single-core PSoC™ 6 MCU devices.

Generally, all memory and peripherals are shared by all bus masters. Shared resources are accessed through standard Arm multi-layer bus arbitration. Exclusive accesses are supported by an inter-processor communication (IPC) block, which implements semaphores and mutual exclusion (mutexes) in hardware.

A dual-core architecture, along with the DMA and cryptography (Crypto) bus masters, presents unique opportunities for system-level design and performance optimization in a single MCU. With two CPUs you can:

- Allocate tasks to CPUs so that multiple tasks may be done at the same time.
- Allocate resources to CPUs so that a CPU may be dedicated to managing those resources, improving efficiency.
- Enable and disable CPUs to minimize power draw.
- Send data between the CPUs using the IPC block. For more information, see the code example [mtb-example-psoc6-dual-cpu-ipc-pipes](#).

In one example application, the Cortex®-M0+ CPU (CM0+) can “own” and manage all communication channels. The Cortex®-M4 CPU (CM4) can send and receive messages from the channels via CM0+. This frees CM4 to do other tasks while CM0+ manages the communication details.

5.10.1.1 Tools and libraries

Infineon provides two development platforms that you can use for application development with PSoC™ 6 MCU:

- **ModusToolbox™:** ModusToolbox™ cross platform software includes configuration tools, low-level drivers, middleware libraries, and operating system support, as well as other packages that enable you to create MCU and wireless applications. It also includes the optional Eclipse IDE for ModusToolbox™.
- **PSoC™ Creator:** A Infineon-proprietary IDE that runs on Windows only. It supports a subset of PSoC™ 6 MCU devices as well as other PSoC™ device families such as PSoC™ 3, PSoC™ 4, and PSoC™ 5LP.

Both platforms provide a mechanism to load and run applications in third-party tools, such as Keil µVision, IAR Embedded Workbench and Visual Studio Code.

Infineon also provides a set of libraries to facilitate application development with PSoC™ 6 MCU:

- **Peripheral Driver Library (PDL):** PDL drivers abstract hardware functions into a set of easy to use APIs. For more information on the PDL, go to the [mtb-pdl-cat1](#) webpage. This library is available in PSoC™ Creator and ModusToolbox™ software.
- **Hardware Abstraction Layer (HAL):** HAL is built on top of the PDL. It provides a high-level interface to configure and use hardware blocks on Infineon MCUs. It is a generic interface that can be used across multiple product families. For more information on the HAL, go to the [mtb-hal-cat1](#) webpage. The HAL is available only in ModusToolbox™ software. PSoC™ Creator does not have support for HAL.
- **Middleware:** This category includes multiple libraries that implement APIs for various domains, capacitive sensing is one such example. A middleware library may be created by Infineon or come from a third party. In PSoC™ Creator, the middleware is usually available as a PSoC™ Creator Component. In ModusToolbox™ software, it is available as a library.
- **Board Support Package (BSP):** BSP provides a standard interface to a board's features and capabilities. The API is consistent across Infineon kits. Other software (such as middleware or an application) can use the BSP to configure and control the hardware. The BSP is available in ModusToolbox™ software only. Although PSoC™ Creator does not offer BSP, you can install the kit software, allowing you to start a new project based on the selected kit.

5 PSoC™ 6 application notes~~DRAFT~~ **5.10.1.2 How to use this document**

This document assumes that you are familiar with PSoC™ 6 MCU architecture, and application development for PSoC™ devices using either ModusToolbox™ software or PSoC™ Creator IDE. For an introduction to PSoC™ 6 MCU, see the following:

- A PSoC™ 6 MCU [device datasheet](#)
- [AN228571](#) - Getting Started with PSoC™ 6 MCU on ModusToolbox™ software
- [AN221774](#) - Getting Started with PSoC™ 6 MCU on PSoC™ Creator
- [AN210781](#) - Getting Started with PSoC™ 6 MCU with Bluetooth® low energy connectivity on PSoC™ Creator

If you are new to ModusToolbox™ software, see the [ModusToolbox™ software home page](#). Some PSoC™ 6 MCU devices are not supported by PSoC™ Creator; in this case, ModusToolbox™ software must be used. If you are using ModusToolbox™ software, make sure that it's version 3.0 or higher for designs based on PSoC™ 6 MCU devices with dual CPUs.

If you are new to PSoC™ Creator, see the [PSoC™ Creator home page](#). Use PSoC™ Creator version 4.3 or higher for PSoC™ 6 MCU-based designs.

Initial sections of this application note cover general concepts for dual-core MCUs and how they are implemented in PSoC™ 6 MCU. To skip to an overview of creating a ModusToolbox™ software or PSoC™ Creator project for a PSoC™ 6 dual-core MCU, go to the [PSoC™ 6 MCU dual-core development](#) section.

~~DRAFT~~

5 PSoC™ 6 application notes

5.10.2 General dual-core concepts

The process of developing firmware for a dual-core MCU is similar to that for a single-core MCU, except that you write code for two CPUs instead of one. You should also consider any need for inter-processor communication.

- **Performance:** The main advantage of having two CPUs is that you essentially multiply your CPU power and bandwidth. How to use that increased bandwidth depends on the tasks that your application must perform.
- **Single task:** A single-task application may be less of a fit for a dual-core MCU unless the application is large and complex. In PSoC™ 6 MCU, you can execute the task on one of the CPUs and put the other CPU to sleep to reduce power.
- **Dual task:** This is the most obvious fit; assign each task to a CPU. Assign the task with larger computing requirements to the higher-performance CPU, i.e., Cortex®-M4 in PSoC™ 6 MCU.
- **Multiple tasks:** Again, assign each task to a CPU. In each CPU, you must include a method for executing each task in a timely fashion.
- **RTOS:** A complex multitasking system may be managed by a real-time operating system (RTOS). An RTOS basically allocates a number of CPU cycles to each task, depending on the task priority or whether a task is waiting for an event. You effectively do that yourself by assigning tasks to the CPUs. Some examples of dual-core RTOS architectures are:
 - Each CPU has its own RTOS and its own set of tasks. Each RTOS should include a task to manage communications with the other CPU.
 - Only one CPU has an RTOS and multiple tasks. The other CPU is idle until it is messaged to do a specified task. It then wakes up and does the task, then messages the result back to the first CPU. As an example, the lower-performance CPU, CM0+ in PSoC™ 6 MCU, can use the higher-performance CPU, CM4 in PSoC™ 6 MCU, to do computation-intensive tasks when needed.
- **Power:** In a dual-core system, firmware can start and stop the CPUs to fine-tune power usage. In the previous example, to reduce power, the high-performance CPU is placed into a sleep state until needed for a computation-intensive task.
- **Debug:** Debugging two bodies of code at the same time may be a complex process. Usually you debug code for one CPU, then debug code for the other CPU. In addition, a device such as an oscilloscope or a logic analyzer may be useful for monitoring communication between the CPUs.

5 PSoC™ 6 application notes

5.10.3 PSoC™ 6 MCU dual-core architecture

~~CONFIDENTIAL~~

Figure 1 shows the overall dual-core architecture in PSoC™ 6 MCU. (For detailed block diagrams of PSoC™ 6 MCU, see the [device datasheet](#)) Specific features and other details related to dual CPUs are listed in this section. For more information, see the [ARM® documentation sets for Cortex® - M4 and Cortex® -M0+](#), and the PSoC™ device [technical reference manual \(TRM\)](#).

- **CPUs:** Both CPUs –Cortex® M4 and Cortex® M0+ – are 32-bit. CM4 runs at up to 150 MHz and has a floating-point unit (FPU). CM0+ runs at up to 100 MHz.
- CM4 is the main CPU. It is designed for a short interrupt response time, high code density, and high throughput. The CM0+ CPU is secondary; it is used in PSoC™ 6 MCU to implement system calls and device-level security, safety, and protection features. CM0+ is also recommended for functions such as BLE communications and CAPSENSE™.
- **Performance:** CM0+ typically operates at a slower clock speed than CM4. The CM0+ instruction set is more limited than that of CM4. Therefore, it may require more cycles to implement a function on CM0+, and the cycle time is longer. Keep this in mind when deciding to which CPU to allocate tasks.
- **Security:** PSoC™ 6 MCU has several security features; see the [TRM](#) for details. To meet security requirements, CM0+ is used as a "secure CPU". It is considered to be a trusted entity; it executes Infineon system code and application code. The use of CM0+ for system and security tasks may limit its availability for applications. For more information, on secure systems, see [AN221111 - PSoC™ 6 MCU designing a custom secured system](#).

Device system calls may be initiated by either CPU but are always executed by CM0+.

- **Startup sequence:** After device reset, only CM0+ executes; CM4 is held in a reset state. CM0+ first executes Infineon system and security code, including SROM code, FlashBoot, and Secure Image. For more information on these code modules, see the Boot Code chapter of the Architecture [TRM](#) for details. After CM0+ executes the system and security code, it executes the application code. In the application code, CM0+ may release the CM4 reset, causing CM4 to start executing its application code. PSoC™ Creator [auto-generates code in CM0+ main\(\)](#) to release the CM4 reset. In ModusToolbox™ software, the BSP chooses and includes one of the several available CM0+ prebuilt images.
- **Inter-processor communication (IPC):** IPC enables the CPUs to communicate and synchronize activities. The IPC hardware contains register structures for IPC channel functions and IPC interrupts. IPC channel registers implement mutual exclusion (mutex) lock/release mechanisms, and messaging between the CPUs. IPC interrupt registers generate interrupts to both CPUs for messaging events and lock/release events.
- **Interrupts:** Each CPU has its own set of interrupts. All peripheral interrupt lines are hard-wired to specific CM4 interrupt inputs. Peripheral interrupts are also multiplexed to CM0+'s limited set of 8 interrupt inputs (32 interrupt inputs in the CY8C61x7, CY8C62x7, and CY8C63x7). See [Interrupt Assignment Considerations](#).
- **Power modes:** PSoC™ 6 MCU has several power modes that can affect either the entire system or just a single CPU. CPU power modes are Active, Sleep, and Deep Sleep as defined by Arm®. Device system power modes are LP, ULP, Deep Sleep, and Hibernate.
 - System Low Power (LP) mode is the default operating mode of the device after reset. It provides the maximum performance. While in System LP mode, the CPUs may operate in any of the Arm®-defined modes.
 - System Ultra Low Power (ULP) mode is identical to LP mode with a performance tradeoff made to achieve a lower system current. This tradeoff lowers the core operating voltage, which then requires a reduced operating clock frequency, and limited high-frequency clock sources. While in system ULP mode, the CPUs may operate in any of the Arm®-defined modes.
 - In System Deep Sleep mode, all high-speed clock sources are OFF. This in turn stops both CPUs and makes high-speed peripherals unusable. However, low-speed clock sources and peripherals continue to operate, if configured and enabled by the firmware. Interrupts from these peripherals cause the

5 PSoC™ 6 application notes

~~DATE~~ device to return to System LP or ULP mode and one or more CPUs to wake up to Active mode. Each CPU has a Wakeup Interrupt Controller (WIC) to wake up the CPU.

- System Hibernate mode is the lowest power mode of the device. It is intended for applications that may go into a dormant state. The device goes through a reset on wakeup from Hibernate. See [Startup sequence](#).
- In CPU Active mode, the CPU executes code and all logic and memory is powered. The device must be in System LP or ULP mode.
- In CPU Sleep mode, the CPU clock is turned OFF and the CPU halts code execution. The device must be in System LP or ULP mode.
- In CPU Deep Sleep mode, the CPU requests the device to go into System Deep Sleep mode. When the device is ready, it enters System Deep Sleep mode. In PSoC™ 6 MCU, both CPUs must enter CPU Deep Sleep before the system transitions to Deep Sleep. If only one CPU has entered CPU Deep Sleep mode, the system remains in LP or ULP mode.
- For more information on PSoC™ 6 MCU power modes, see [AN219528 - PSoC™ 6 MCU low-power modes and power reduction techniques](#).
- **Debug:** PSoC™ 6 MCU has a Debug Access Port (DAP) that acts as the interface for device programming and debug. An external programmer or debugger (the "host") communicates with the DAP through the device Serial Wire Debug (SWD) or Joint Test Action Group (JTAG) interface pins. Through the DAP (and subject to device security restrictions), the host can access the device memory and peripherals as well as the registers in both CPUs.
- Each CPU offers several debug and trace features as follows:
 - CM4 supports six hardware breakpoints and four watchpoints, 4-bit embedded trace macrocell (ETM), serial wire viewer (SWV), and printf()-style debugging through the single-wire output (SWO) pin.
 - CM0+ supports four hardware breakpoints and two watchpoints, and a micro trace buffer (MTB) with 4 KB dedicated RAM.

PSoC™ 6 MCU also has an [Embedded Cross Trigger](#) for synchronized debugging and tracing of both CPUs.

Some third-party IDEs support synchronized dual-core debugging; see [Debug considerations](#). Infineon's ModusToolbox™ software supports dual-core debugging (unsynchronized), and PSoC™ Creator supports debugging a single CPU (either CM4 or CM0+) at a time.

~~5 PSoC™ 6 application notes~~

~~5.10.4 PSoC™ 6 MCU dual-core development~~

This section shows only those development aspects that are unique to PSoC™ 6 MCU dual-core devices. To learn more about PSoC™ 6 MCU, ModusToolbox™ software, or PSoC™ Creator, see one or more of the following:

- [AN228571](#) - Getting Started with PSoC™ 6 MCU on ModusToolbox™ software
- [AN221774](#) - Getting Started with PSoC™ 6 MCU on PSoC™ Creator
- [AN210781](#) - Getting Started with PSoC™ 6 MCU with Bluetooth® low energy connectivity on PSoC™ Creator
- The [ModusToolbox™ software home page](#). Some PSoC™ 6 MCU devices are not supported by PSoC™ Creator; in this case, ModusToolbox™ software must be used. If you are using ModusToolbox™ software, then make sure it's version 3.0 or higher for designs based on PSoC™ 6 MCU devices with dual CPUs
- The [PSoC™ Creator home page](#). Use PSoC™ Creator version 4.3 or higher for PSoC™ 6 MCU-based designs

5.10.4.1 ModusToolbox™ software instructions

ModusToolbox™ software application development for a PSoC™ 6 MCU dual-core device is similar to that for any other device supported by ModusToolbox™ software. By default, the Board Support Packages (BSP) that come with the ModusToolbox™ software provides several prebuilt CM0+ application images. This section explains how to create your own CM0+ application as a substitute for the prebuilt images.

To create a dual-core application, you can follow one of the following methods:

1. If you are starting dual-core application development from scratch, you can create your new application based on an existing dual-CPU code example hosted [here](#). You can use any code example with a repo name that includes “dual-CPU” as a start point. For most cases, the [mtb-example-psoc6-dual-cpu-empty-app](#) is a good choice.
2. If you have an existing single-core application that you want to convert to a dual-core application, follow the steps from [Creating a dual-core application](#) section.

5.10.4.1.1 Creating a dual-core application

This section shows how to create a dual-core application step-by-step by using an existing single-core application and a starter dual-core application. Here the starter dual-core application will act as a template defining the dual-core application structure. The existing single-core application's contents can easily be copied into the starter dual-core application.

For the single-core application, as an example, the [mtb-example-psoc6-hello-world](#) is chosen. For the starter dual-core application, the [mtb-example-psoc6-dual-cpu-empty-app](#) is chosen. The CY8CPROTO-062-4343W BSP is chosen for both applications.

The applications can be created using one of the following methods:

- [Eclipse IDE for ModusToolbox™](#)
- [Command-line Interface \(CLI\)](#)

Eclipse IDE for ModusToolbox™

1. Launch Eclipse IDE for ModusToolbox™ 3.0 or later and select a workspace.
2. From the **Quick Panel**, click **New Application**.
3. Once the project creator wizard opens, expand **PSoC™ 6 BSPs** and select **CY8CPROTO-062-4343W**. Click **Next**.
4. In the new window, expand **Getting Started**, select **Dual-CPU Empty PSoC6 App** and **Hello World**. Click **Create**.
5. In [Figure 405](#), you see both applications created in the workspace.

5 PSoC™ 6 application notes

DRAFT

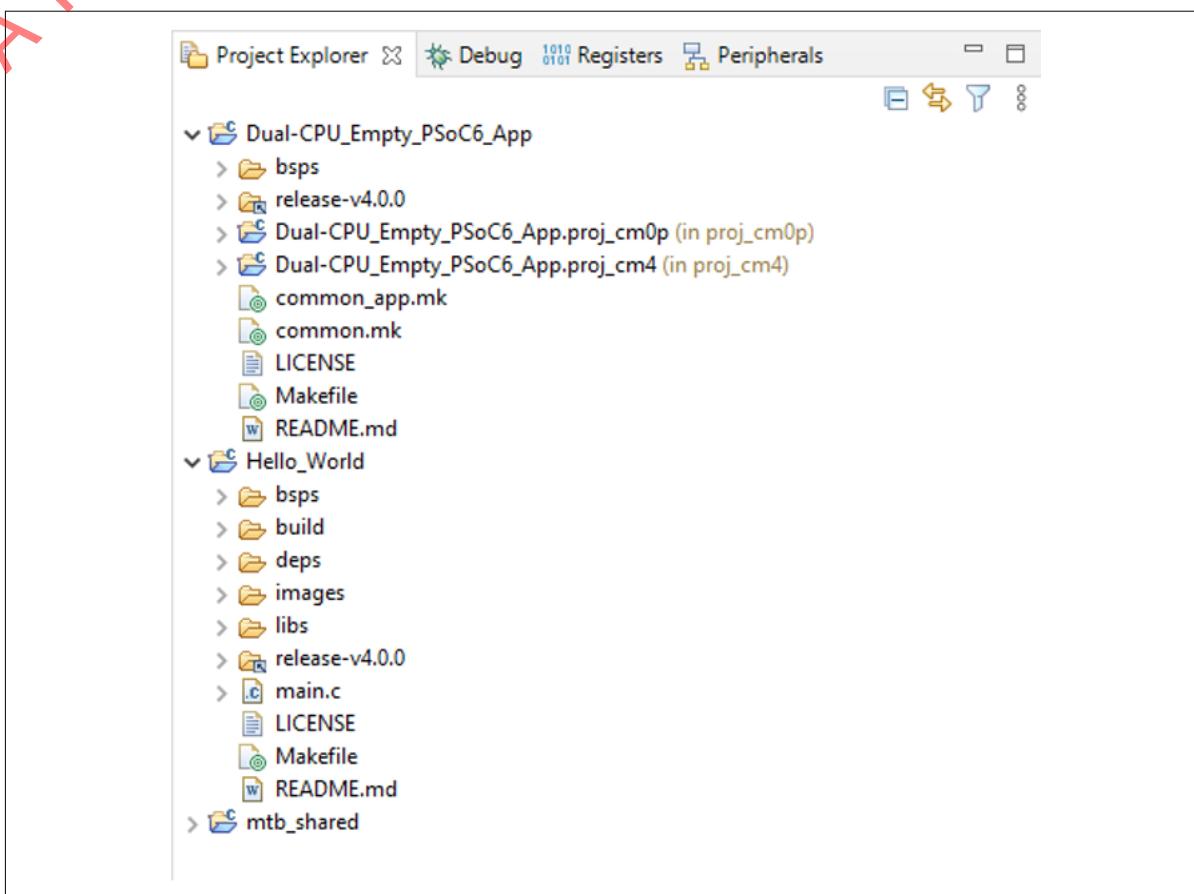


Figure 405 Workspace in ModusToolbox™ containing both single-core and dual-core applications

Command-line Interface (CLI)

1. For project creation, open a CLI terminal and navigate to ModusToolbox™ software installation folder such as <install_path>/ModusToolbox/tools_<version>/project-creator/.
2. Create both the applications by executing the following project-creator-cli commands.

- ```
project-creator-cli \
--board-id CY8CPROTO-062-4343W \
--app-id mtb-example-psoc6-hello-world \
--user-app-name Hello_World \
--target-dir "C:/workspace_directory"
```
- ```
project-creator-cli \
--board-id CY8CPROTO-062-4343W \
--app-id mtb-example-psoc6-dual-cpu-empty-app \
--user-app-name Dual-CPU_Empty_PSoC6_App \
--target-dir "C:/workspace_directory"
```

Alternatively, you can execute the **project-creator** tool in the same folder and follow the same steps described in [Eclipse IDE for ModusToolbox™](#).

For more information, see Section 2.3 in the [ModusToolbox™ user guide](#).

~~5 PSoC™ 6 application notes~~

~~Modify application contents~~

The following contents from the Hello World example need to be selectively copied to the CM4 project (`proj_cm4`) of the Dual CPU Empty application:

- ~~1.~~ **Makefile variables** – Open the Makefile of the Hello World application and `proj_cm4` application side-by-side. Compare and copy the variable values from the Hello World Makefile to the `proj_cm4` Makefile. If some of the variables defined in the Hello World Makefile are not present in the `proj_cm4` Makefile, they should be available in the `common.mk` and `common_app.mk` Makefiles at the root level of the dual-core empty application.

Current values of variables such as `MTB_TYPE`, `APPNAME`, `COMPONENTS`, and `DISABLE_COMPONENTS` in the Dual CPU Empty application Makefiles should not be deleted or modified. Adding new values is still allowed for variables such as `COMPONENTS` and `DISABLE_COMPONENTS`.

For the Hello World application case, there are no Makefile variables that need to be copied.

- ~~2.~~ **Source files** – Replace all the source files from the `proj_cm4` application with the Hello World application.
- ~~3.~~ **Libraries/Middleware** – The libraries or middleware used in the Hello World application should also be added to the `proj_cm4` application. Follow these steps to add them:
 - a. In the Eclipse IDE of ModusToolbox™, from the quick planel launch **Library Manager** tool. Alternatively, you can launch the Library Manager from the tools folder of ModusToolbox™ installation directory, in this case the correct path to the Dual CPU Empty application needs to be selected from the Library Manager window.
 - b. Once the **Library Manager** opens, click **Add Library**.
 - c. Select **Target Project** as `proj_cm4`.
 - d. Under **Peripheral**, select **retarget-io** library, and click **OK**.
 - e. Click **Update** from the **Library Manager** window. Wait for the libraries to get added; see the log messages for progress.
 - f. Once the libraries are updated, click **Close**.

The CM0+ project (`proj_cm0p`) of the Dual CPU Empty application enables the CM4 core and goes to Deep Sleep. To use the core for any other purpose, the `cybsp_init()` function needs to be called from the CM0+ core. This function will do all the system-level initialization required to run peripherals on the CM0+ core. Modify the `main.c` file of `proj_cm0p` as shown in [Code Listing 1](#).

5 PSoC™ 6 application notes

Code Listing 1

```
#include "cy_pdl.h"
#include "cycfg.h"
#include "cybsp.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 global interrupts */
    __enable_irq();

    /* Enable CM4. CY_CORTEX_M4_APPL_ADDR must be updated if CM4 memory layout is changed. */
    Cy_SysEnableCM4(CY_CORTEX_M4_APPL_ADDR);

    for (;;)
    {
        Cy_SysPm_DeepSleep(CY_SYSPM_WAIT_FOR_INTERRUPT);
    }
}
```

The Infineon HAL is not yet designed to run simultaneously on CM0+ and CM4 core. Since the proj_cm4 already uses HAL, proj_cm0p should only use PDL APIs.

Build and program

The application can be built and programmed using one of the following methods:

- Using Eclipse IDE for ModusToolbox™
 1. Select the **Dual-CPU_Empty_PSoC6_App** application from the project explorer.
 2. From the quick panel, select **Dual-CPU_Empty_PSoC6_App Program MultiCore (KitProg3)**. This will build and program both projects – proj_cm0p and proj_cm4, one after the other.
 3. Alternatively, you can program individual projects (proj_cm0p or proj_cm4), but this is only possible through CLI interface as explained below.
- Using Command-line Interface (CLI)
 1. Open a modus-shell and navigate it to the root directory of the **Dual-CPU_Empty_PSoC6_App** application.
 2. Run command -`make program`. This will build and program both projects – proj_cm0p and proj_cm4, one after the other.
 3. Alternatively, you can program individual projects (proj_cm0p or proj_cm4). Navigate the modus-shell to the required project directory. Run command -`make program_proj`.

~~5 PSoC™ 6 application notes~~

The build artifacts for each project is generated in its own project directory under a folder named `build`. Note that the `build` folder in the application level has a copy of the project hex files. It also has a combined hex file containing the data of individual project hex files.

If the projects fail to build due to a flash overflow error, follow the steps from the [Customizing linker scripts](#) section to allocate the required memory for each project.

5.10.4.1.2 Customizing linker scripts

The CM0+ and CM4 projects each have their own linker scripts supplied by the BSP. By default, the CM0+ CPU consumes only 8192 [0x2000] bytes of flash and 8192 [0x2000] bytes of SRAM. If your CM0+ project requires more memory, both linker scripts require changes. Note that the increment/decrement in flash size should be in multiples of flash row size (512 bytes). The following steps list the changes:

1. The CM0+ linker script is located at directory - `bsps/TARGET_APP_<BSP_NAME>/COMPONENT_CM0P/`
By default, the BSP supports the ARM (*.sct), GCC_ARM (*.ld), and IAR (*.icf) toolchains. Each toolchain has its own linker script.
2. Edit the FLASH and SRAM size as desired:

Toolchain:	Where to change
ARM (*.sct)	#define RAM_SIZE 0x0000 2000 #define FLASH_SIZE 0x0000 2000
GCC_ARM (*.ld)	ram (rwx) : ORIGIN = 0x08000000, LENGTH = 0x 2000 flash (rx) : ORIGIN = 0x10000000, LENGTH = 0x 2000
IAR (*.icf)	define symbol __ICFEDIT_region_IRAM1_end__ = 0x0800 1FFF ; define symbol __ICFEDIT_region_IROM1_end__ = 0x1000 1FFF ;

3. The CM4 linker script is located at directory - `bsps/TARGET_APP_<BSP_NAME>/COMPONENT_CM4/`
4. Edit the values based on the CM0+ memory size:

Toolchain:	Where to change
ARM (*.sct)	#define RAM_START 0x0800 2000 #define RAM_SIZE 0x000 FD800 #define FLASH_CM0P_SIZE 0x 2000
GCC_ARM (*.ld)	FLASH_CM0P_SIZE = 0x 2000 ; ram (rwx) : ORIGIN = 0x0800 2000 , LENGTH = 0x FD800
IAR (*.icf)	define symbol __ICFEDIT_region_IRAM1_start__ = 0x0800 2000 ; define symbol __ICFEDIT_region_IRAM1_end__ = 0x080 FF7FF ;

5. In the CM0+ Makefile, add to DEFINES the CM4 application address (`CY_CORTEX_M4_APPL_ADDR`). For example, if the size of CM0+ image is 0x8000, set it to:

```
DEFINES=CY_CORTEX_M4_APPL_ADDR=0x10008000
```

5.10.4.1.3 Sharing libraries and peripherals

The `deps` folder of each project contains the list of libraries used by that project. As described earlier ModusToolbox™ provides a tool to manage the libraries and BSPs, called Library Manager. You can launch this tool from the Eclipse IDE for ModusToolbox™, **Quick Panel > Tools > Library Manager**. Alternatively, you can execute the **library-manager** tool located at this folder: (`<install_path>/ModusToolbox/tools_<version>/library-manager/`).

The BSPs added through the library manager by default will be downloaded to a folder named `bsps` located at the root directory of the dual CPU application. Both projects share the BSPs in an application.

5 PSoC™ 6 application notes

The libraries added through the library manager by default will be downloaded to a folder named `mtb_shared`, located one level above the root directory of the dual CPU application. The libraries can be shared or project-specific. When adding the libraries through the library manager, an option is provided to select the project it belongs to. If the same library is added to both the projects, then both the projects can access the same library copy from the `mtb_shared` directory.

If a peripheral is shared, you should only initialize it in one of the CPUs and use some sort of mutex or synchronization mechanism to avoid both CPUs using it at the same time. It is not recommended to use the HAL for shared peripherals, since the CM0+ would not have access to the HAL object created by the CM4.

5 PSoC™ 6 application notes

5.10.4.2 PSoC™ Creator instructions

PSoC™ Creator project development for a PSoC™ 6 MCU dual-core device is similar to that for any other device supported by PSoC™ Creator. To create a new project, select **File > New > Project**. A **Create Project** dialog is displayed, as shown in [Figure 406](#).

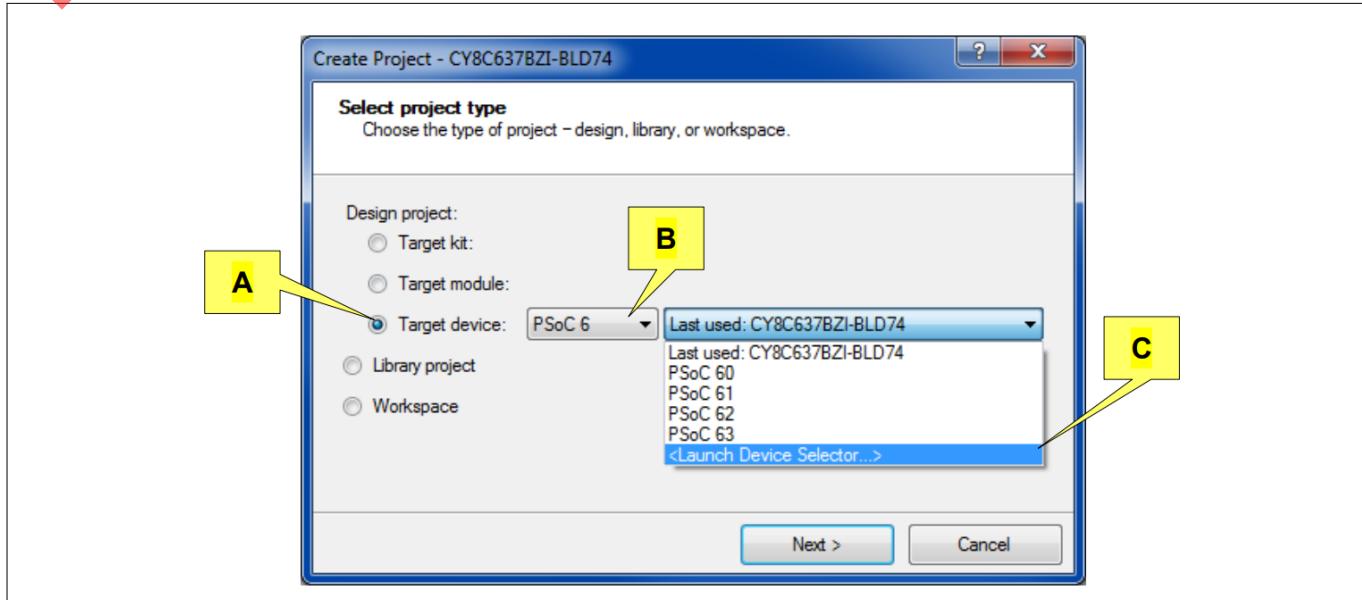


Figure 406 PSoC™ Creator create project dialog

Select **Target Device** (A), and **PSoC 6** (B). On the pull-down list (C), select **<Launch Device Selector...>** to see a list of PSoC™ 6 devices.

[Figure 407](#) shows the Device Selector dialog. To see a list of dual-core devices, click the **CPU** category (D) and select only **CortexM0p, CortexM4**.

In the PSoC™ 6 BLE pioneer kit [CY8CKIT -062- BLE](#), the PSoC™ 6 MCU dual-core device part number is CY8C6347BZI-BLD53. In the PSoC™ 6 Wi-Fi-BT pioneer kit [CY8CKIT -062- WiFi -BT](#), the PSoC™ 6 MCU dual-core device part number is CY8C6247BZI-D54.

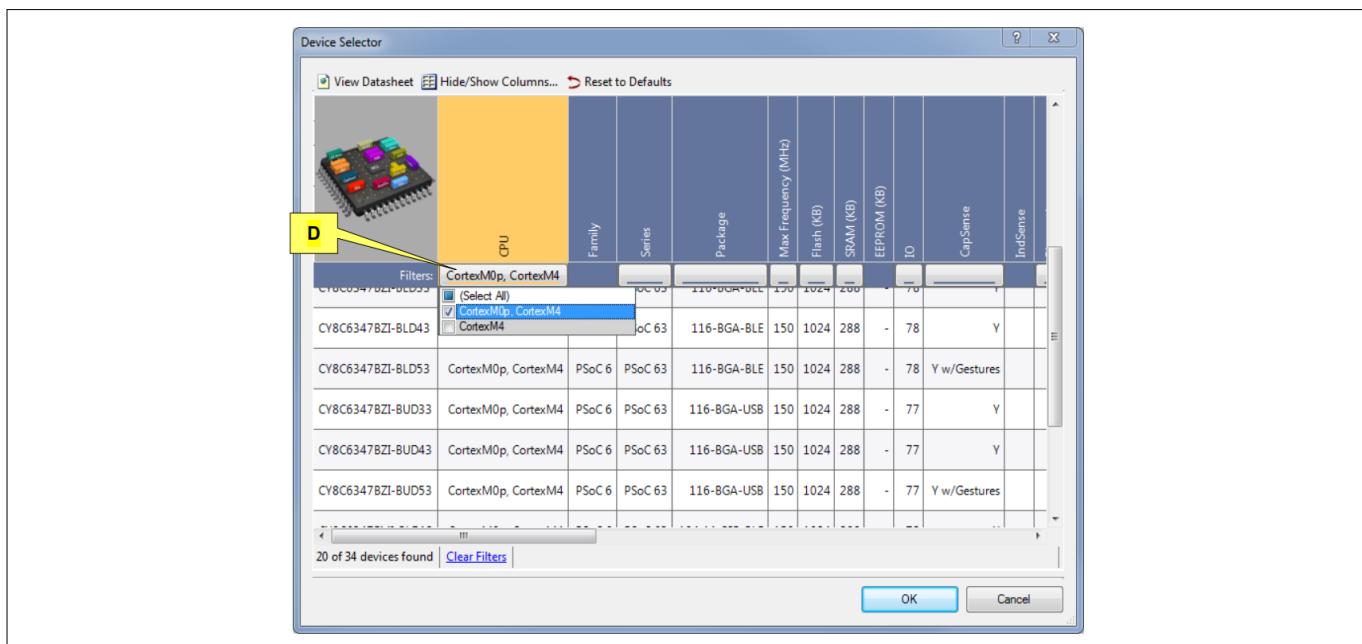


Figure 407 PSoC™ Creator device selector dialog

~~5 PSoC™ 6 application notes~~

After selecting a PSoC™ 6 MCU part, the rest of the project creation process is the same as for other devices. Click through the rest of the **Create Project** dialogs; PSoC™ Creator creates the project.

The initial project windows layout (Figure 408) includes a **Workspace Explorer** window with the following features for dual-core devices:

- Separate `main.c` files – `main_cm0p.c` and `main_cm4.c` – for each CPU. Sources in the folders CM0p (Core 0) and CM4 (Core 1) are compiled into separate binaries for the respective CPUs.
- A Shared Files folder. Source files in this folder are compiled into both binaries.

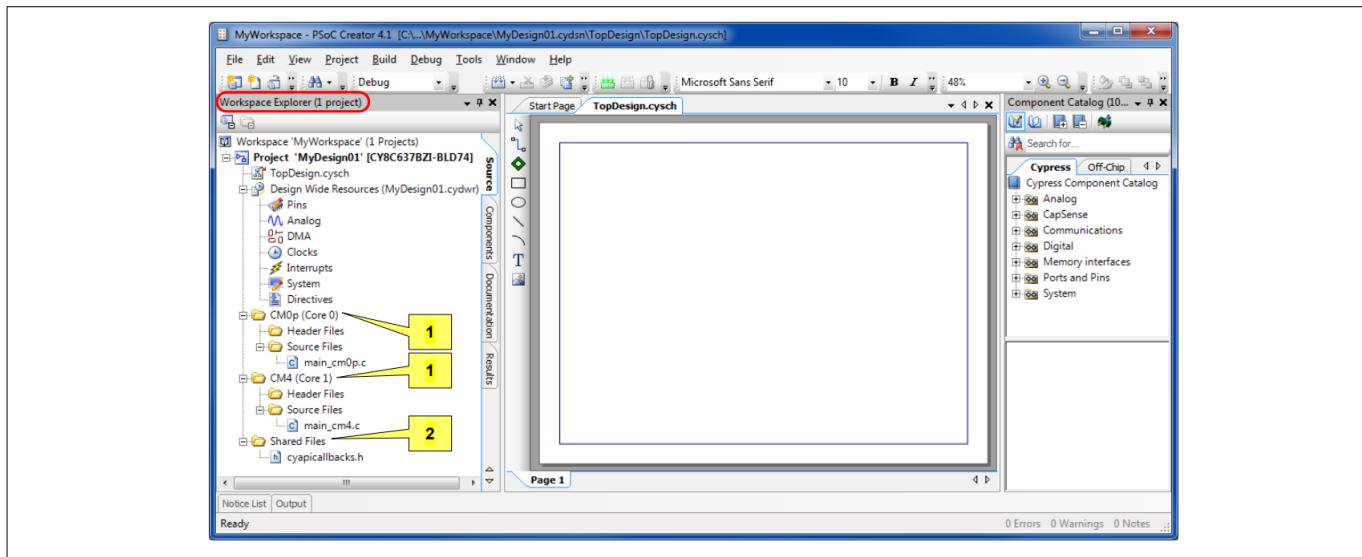


Figure 408 PSoC™ Creator initial project layout for dual-core devices

The initial project layout also includes a TopDesign hardware schematic, along with an associated component catalog window.

After the project is created, implement your hardware design by dragging Components onto the schematic, and configuring and wiring them.

When schematic design entry is complete, select **Build > Generate Application**. This creates several system source code files and folders in the existing folders as well as in the new folder Generated Source, as Figure 409 shows.

The generated source contains drivers for each Component on the schematic, as well as CYPRESS™ peripheral driver library (PDL). The PDL is a software development kit (SDK) that integrates device header files, start-up code, and peripheral drivers. The peripheral drivers abstract the hardware functions into a set of easy-to-use APIs.

For more information on the PDL, select PSoC™ Creator **Help > Documentation > Peripheral Driver Library**. Also, each Component has a datasheet that documents the driver API for that Component. Right-click the Component and select **Open Datasheet....**

PSoC™ Creator IDE creates several other files and folders, and places them in existing folders CM0p (Core 0), CM4 (Core 1), and Shared Files. These files generally support configuration, start-up, and linking options for PSoC™ Creator as well as other IDEs. For more information on these files, see PSoC™ Creator Help article, **Generated Files (PSoC™ 6)**.

5 PSoC™ 6 application notes

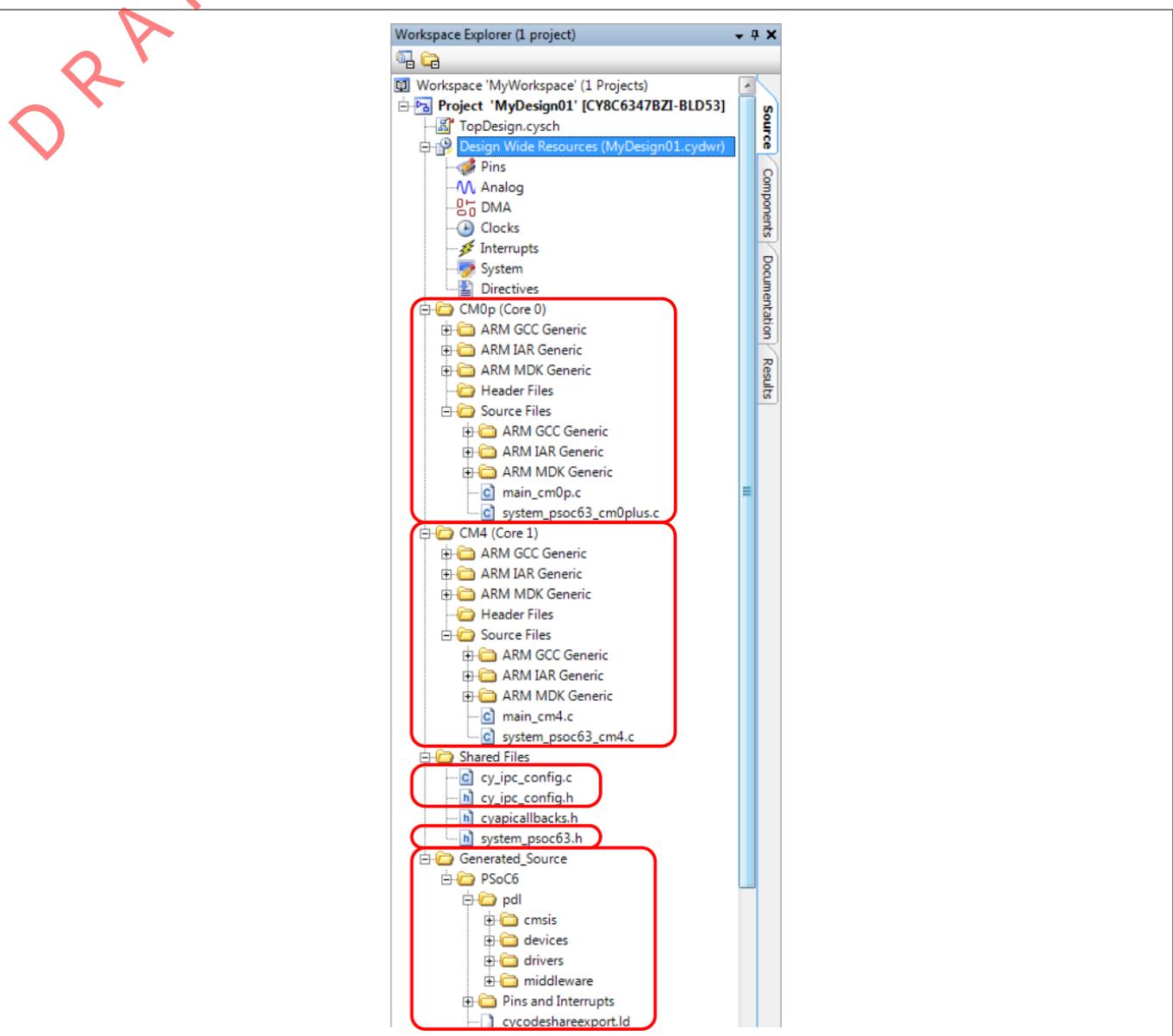


Figure 409 Add generated source to a project

~~5 PSoC™ 6 application notes~~

~~5.10.4.3 Resource assignment considerations~~

All PDL driver source and other API files are available to both CPUs. If code for a CPU references any API element in a generated source file, that file is compiled into the binary for that CPU. The same file can be compiled into both binaries – see code example [CE216795 , PSoC™ 6 MCU dual-CPU basics](#).

Note: *Customers using PSoC™ Creator should download the required examples inside the tool using the project creation wizard. The PSoC™ Creator code example links provided in this document point only to the code example document.*

If the same source file is compiled into both binaries, a function in that file is duplicated in both binaries. Even though the copies are in separate builds and binaries, in some cases it is convenient to consider a function to be executed simultaneously by both CPUs.

As noted [previously](#), it is possible for a peripheral to be accessed by both CPUs; for example, both CPUs may send data through the same UART.

There are two ways to do resource management:

- **Dedicate a resource to one CPU.** Include code to use the resource only in the firmware for the desired CPU. Also, if you are using PSoC™ Creator, a good practice is to indicate on the project schematic the CPU that “owns” the resource, as [Figure 410](#) shows

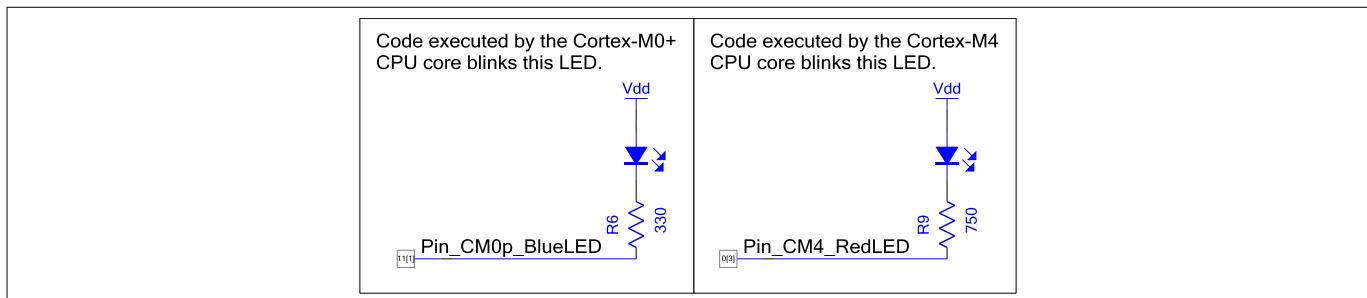


Figure 410 PSoC™ Creator project schematic for dual CPUs controlling separate pin components

- **Share resources between the CPUs.** Code example [CE216795](#) shows how the PSoC™ 6 MCU’s IPC block may be used to implement a mutex to share memory between the CPUs. Use the same technique to share a peripheral resource such as a UART

Flash and SRAM that are allocated in a CPU’s binary are generally separate from that for the other CPU. If custom sections and section placement are defined in the CPUs’ linker scripts, you must ensure that the sections do not overlap. Conversely, another way to share memory is to define for each CPU custom sections with the same address.

5.10.4.4 Power mode transition considerations

Resources such as system clocks are always shared between CPUs and all peripherals. The `cybsp_init()` function calls `cybsp_register_sysclk_pm_callback()` function. This function registers a power management callback that prepares the clock system for Deep Sleep mode and restores the clocks upon wakeup from a Deep Sleep.

The default `cybsp_register_sysclk_pm_callback()` is designed for single CPU examples, where CM0+ is in a Deep Sleep. If the CM4 CPU is also put to Deep Sleep, the callback function disables the clock system. This will not cause any problem to the CM0+ CPU as it is already in a Deep Sleep. In the case of dual CPU examples, this is not the case. If one of the CPUs goes to Deep Sleep, the callback doesn’t check the state of the other CPUs and disables the clock system. This causes the other active CPUs to misbehave.

To avoid the above issue, a custom power management callback function is needed. In this custom callback, you can decide whether to disable the system clocks or not. You can also implement any other custom

~~5 PSoC™ 6 application notes~~

functions required for your application when performing a power mode transition. See the default callback function implementation to write your custom callback in the correct format.

Follow these steps to create and call a custom power management callback function:

1. From the root of your dual CPU example, create a shared folder to add source files.
For example: shared/source/
2. Create a C file with any name.
For example: shared/source/custom_pm_callback.c
3. Open the file and write a custom callback function with the exact same definition and name as shown below:

```
cy_rslt_t cybsp_register_custom_sysclk_pm_callback(void)
{
    . . .
}
```

4. Open the Makefiles of both CM0p and CM4 projects. Make the following changes:
 - a. Edit the SOURCES variable to add the newly created C file to the build process.
SOURCES= ../shared/source/custom_pm_callback.c
 - b. Add CYBSP_CUSTOM_SYSCLK_PM_CALLBACK macro to the DEFINES variable.

5.10.4.5 IPC configuration considerations

Inter-Process Communication (IPC) provides the functionality for multiple CPUs to communicate and synchronize their activities. Following are some of the guidelines one needs to consider while using IPC:

1. The shared memory used for IPC communication can be placed in an unprotected public RAM region. This public_ram region is defined and commented in the default cm0+ linker script provided by the target BSP. To use this public RAM region, uncomment the public_ram line and .cy_sharedmem section in the CM0+ linker script.
2. The public_ram region must be given executable privileges to allow for flash write functions to run out of the SRAM. This can be achieved by changing the rw parameter to rwx parameter in the public_ram line of the CM0+ linker script.
3. If the public_ram region is not used, the shared memory will be allocated in the SRAM region defined for the CM0+ CPU.
4. When using protection units, the user application should take care not to protect the shared memory region.

5.10.4.6 Interrupt assignment considerations

An important consideration for dual-core designs is assigning and handling interrupts. As noted previously, all device interrupts are available to CM4, and a subset of interrupts are routed through multiplexers to CM0+. You must decide which CPU will handle each interrupt.

For more information, see application note [AN217666 - PSoC™ 6 MCU interrupts](#).

ModusToolbox™ software: In ModusToolbox™ software, interrupts are assigned programmatically; at this time, there is no GUI support. Examples of the required code are in [AN217666](#); and in the PDL/HAL documentation, Infineon BLE Middleware Library section, Configure BLESS interrupt subsection. The code in this subsection shows how to assign PSoC™ 6 MCU BLE subsystem (BLESS) interrupts to either CM0+ or CM4. The code can be easily modified to support other interrupt sources. If you are using Infineon HAL, all interrupts required by a HAL driver are assigned to the CPU calling the HAL functions.

5 PSoC™ 6 application notes

DRAFT

Note: In CY8C61x6/7, CY8C62x6/7, and CY8C63xx devices, the CM0+ CPU is only capable of handling 32 system interrupts, out of which only 8 are deep sleep capable. If HAL functions are being used to initialize peripherals which require interrupts, the function will return “none free error” when all interrupt lines are fully utilized.

PSoC™ Creator: For PSoC™ Creator, let us assign interrupts in an example design. Figure 411 shows a design with two interrupts; one from a PWM Component, connected to an Interrupt Component MyPWM_Int; and the other from an I2C Component.

In the Design wide Resources window (file type .cydwr), select the **Interrupts** tab to see all of the interrupts in the design, as Figure 412 shows.

In this example, the I2C component has an interrupt embedded in it. That interrupt is not shown on the schematic in Figure 411; it is shown in the Design-Wide Resources window as MyI2C_SCB_IRQ.

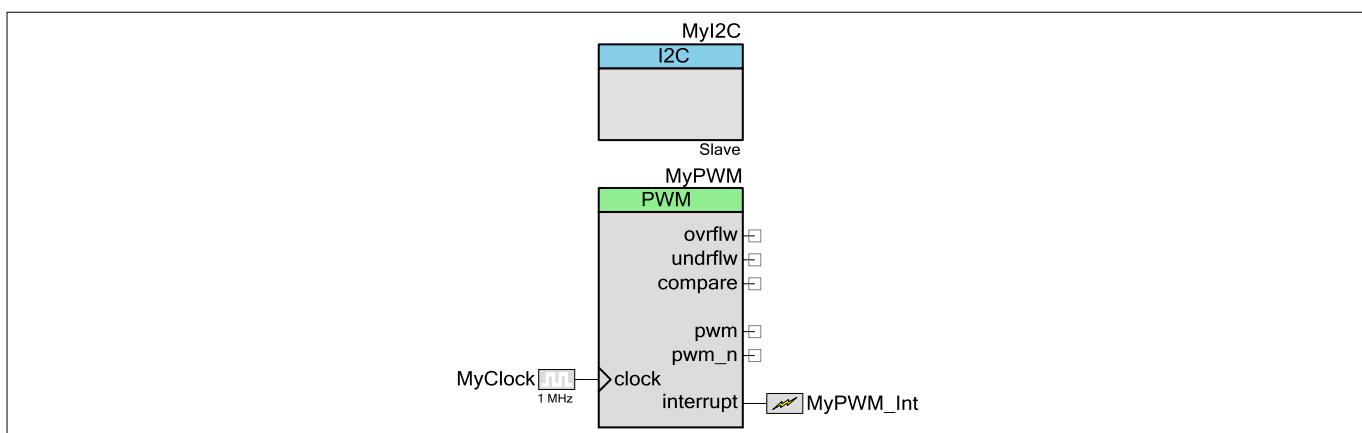


Figure 411 Example schematic design with two interrupts

Check or uncheck the boxes in the **Arm® CM0+ Enable** and **Arm® CM4 Enable** columns to assign interrupts to the respective CPUs.

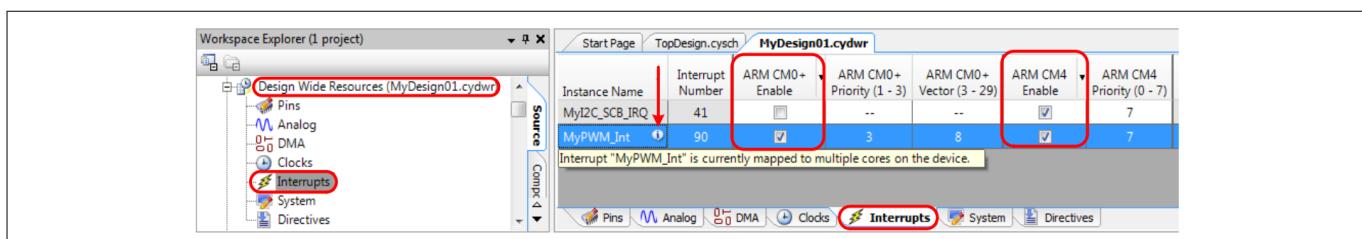


Figure 412 Assign interrupts to the CPUs

Each peripheral interrupt is hard-wired to CM4, so the **Interrupt Number** is automatically assigned by PSoC™ Creator when you build the project. Because interrupts are routed through multiplexers to CM0+, you can select an **Arm® CM0+ Vector** for each interrupt.

Note: A warning symbol and tooltip are displayed if an interrupt is assigned to both CPUs. This is generally not recommended; however, an interrupt can be used to wake up one or both CPUs from their [Sleep modes](#).

5.10.4.7 Debug considerations

Third-party IDEs such as Keil µVision and IAR Embedded Workbench support dual-core debugging. Eclipse IDE for ModusToolbox™ software supports dual-core debugging (unsynchronized), and PSoC™ Creator supports debugging a single CPU (either CM4 or CM0+) at a time.

~~5 PSoC™ 6 application notes~~

~~5.10.4.7.1 ModusToolbox™ instructions~~

Eclipse IDE for ModusToolbox™ software supports asynchronous dual core debugging. You can launch the debug session for both cores simultaneously, but debug activities (For example, halt, stop, breakpoints) on one core will not affect the other. For detailed instructions, open the [ModusToolbox™ User Guide](#), and see the subsection [PSoC™ 6 MCU Programming/Debugging](#).

~~5.10.4.7.2 PSoC™ Creator instructions~~

PSoC™ Creator supports debugging just one CPU at a time. Before starting a debug session with PSoC™ 6 MCU, select the desired debug target (**Debug > Select Debug Target...**), as [Figure 413](#) shows. Select the desired CPU and click **OK/Connect**. To debug the other CPU, you must exit the debugger and then re-enter it with a connection to that CPU.

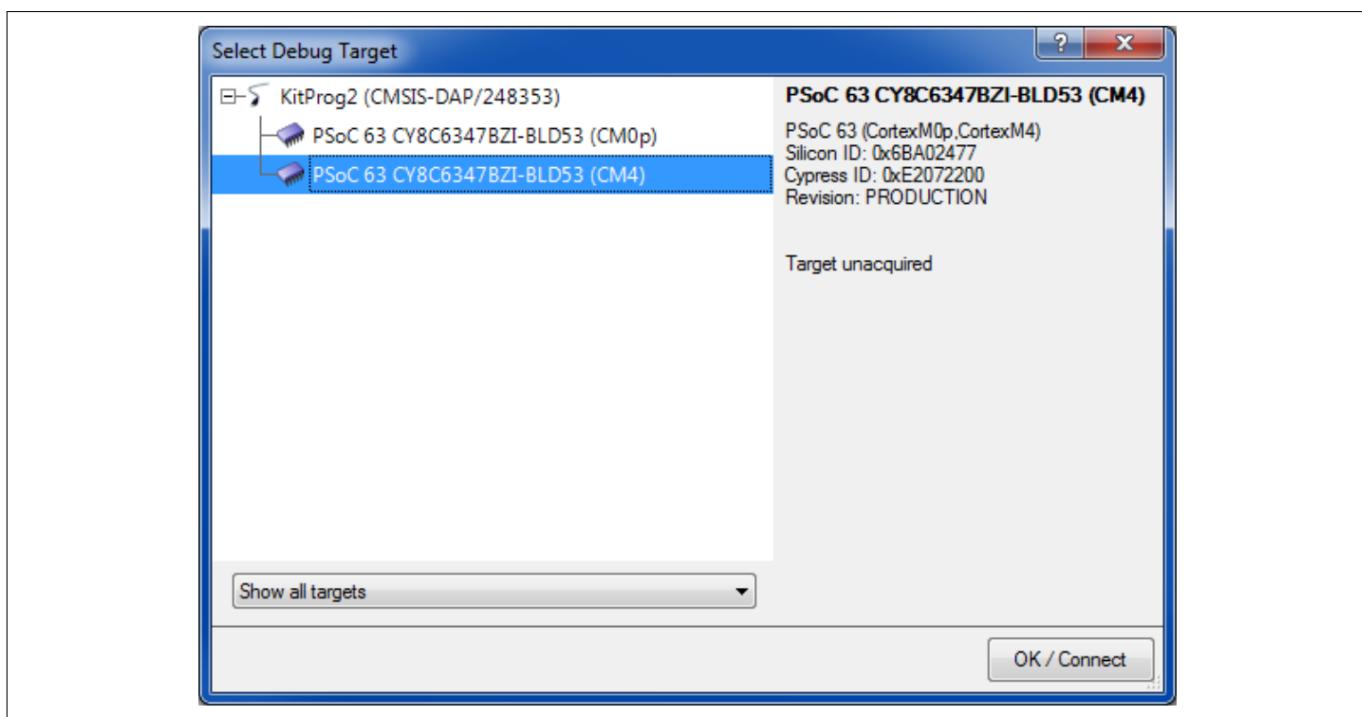


Figure 413 **PSoC™ Creator select CPU for debug**

Recommended: Develop and debug first the portions of code where the CPUs communicate with each other. After that, code executed by an individual CPU can be debugged separately. For example, when the shared memory project in [CE216795](#) was developed, the portion where CM0+ sends an initial message to CM4 was developed and debugged before subsequent portions of code were developed.

You can debug both CPUs simultaneously by using other IDEs such as µVision or IAR. To do so, you must export your PSoC™ Creator project to the other IDE. PSoC™ Creator documents this topic in the help articles [Integrating into 3rd Party IDEs](#), [PSoC™ 6 Designs](#). Review the instructions in the help articles; the general steps are summarized in the following sections.

5.10.4.7.3 Instructions for other IDEs

Exporting ModusToolbox™ applications to supported IDEs

ModusToolbox™ software has a mechanism to load and run applications on the following IDEs:

- IAR Embedded Workbench

5 PSoC™ 6 application notes

- Keil µVision
- Visual Studio Code

For more details on how to export and set up debugging with these tools, see the “Exporting to IDEs” chapter in the [ModusToolbox™ user guide](#).

Exporting from PSoC™ Creator

If you are using PSoC™ Creator, you can export your project to Keil µVision or IAR Embedded Workbench for dual-core debugging. Following are the steps to do so:

1. Configure the PSoC™ Creator Project

2. Create µVision Projects
3. Debug µVision Projects
4. Create IAR-EW Projects
5. Debug IAR-EW Projects

1. Configure the PSoC™ Creator Project

Update the **Target IDEs** settings in the project Build Settings, as [Figure 414](#) shows

For µVision, select **CMSIS Pack: > Generate**. Enter appropriate identifying text for the CMSIS pack in the **Vendor**, **Pack**, and **Version** fields

Recommended: select **Toolchain: > ARM MDK Generic**

For IAR, you only need to select **IAR EW-ARM: > Generate**. (An advanced option, **Generate without copying PDL files**, is also available.) IAR has its own compiler (not supported by PSoC™ Creator), so the Toolchain selection is irrelevant

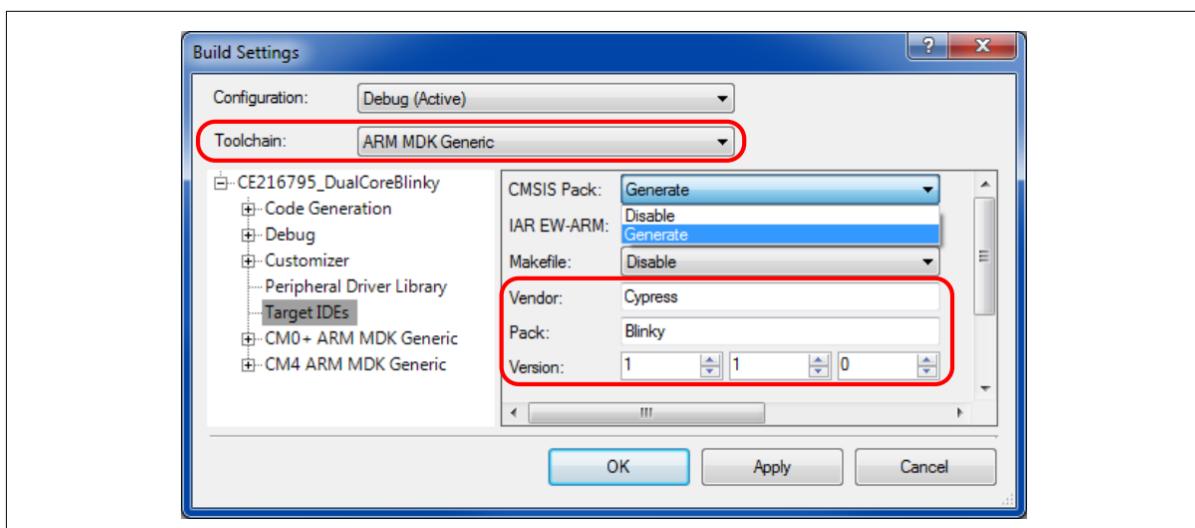


Figure 414 Build settings for target IDEs

Then build your PSoC™ Creator project in the usual manner. A folder Export is created in your <project>.cydsn folder, which contains relevant files for exporting to the selected IDE or IDEs

For µVision, after the PSoC™ Creator project is built, find the corresponding .pack file in the folder Export \ Pack. Double-click the file to install it as a µVision pack, as [Figure 415](#) shows.

Note: *Do not use the µVision Pack Installer Wizard File Import function to install this pack.*

5 PSoC™ 6 application notes

DRAFT

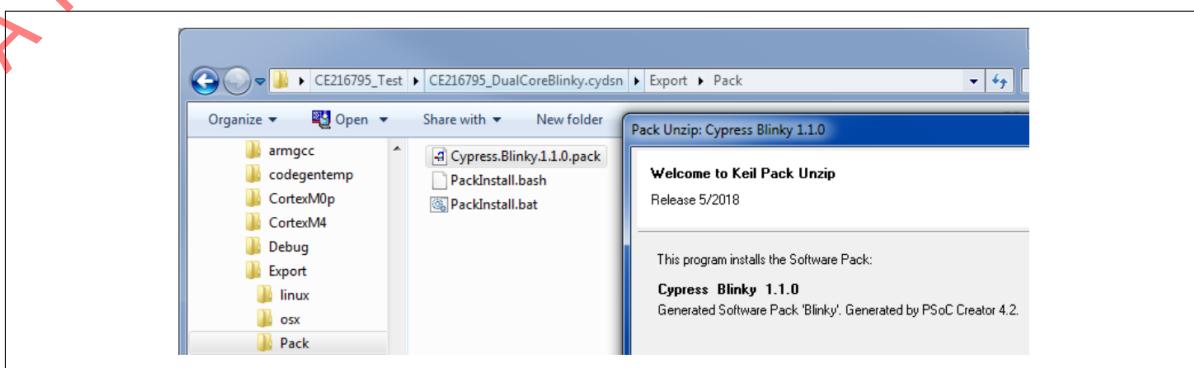


Figure 415 Install µVision pack from PSoC™ Creator project

if you update the PSoC™ Creator project, consider changing the µVision pack version number (see [Figure 414](#)) and installing the new pack

For more information, see [AN219434 - Importing PSoC™ Creator Code into an IDE for a PSoC™ 6 Project](#)

2. Create µVision projects

For µVision, you must create two projects: one for each PSoC™ 6 MCU CPU: CM0+ and CM4. Do the following:

Recommended: create a new folder (For example, `µvisionBuild`) within your PSoC™ Creator <project>.cydsn folder to store all µVision project files separately from the PSoC™ Creator files (this is different from the [IAR instructions](#)). Within that folder, create another new folder for CM4 object files (For example, `ObjectsM4`), as [Figure 416](#) shows:

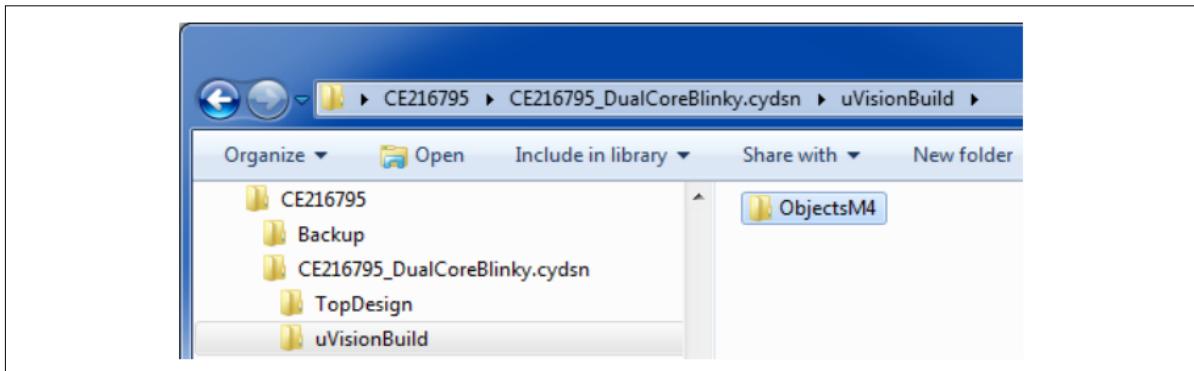


Figure 416 New folders for µVision projects

Open µVision 5.25 or later and create a new project (**Project > New µVision Project...**) in the `µvisionBuild` folder

Recommended: name the project based on the original PSoC™ Creator project name and the target CPU. For example, for the [CE216795](#) dual-core blinky project, create a µVision project `BlinkyM0p` for the CM0+ CPU, as [Figure 417](#) shows:

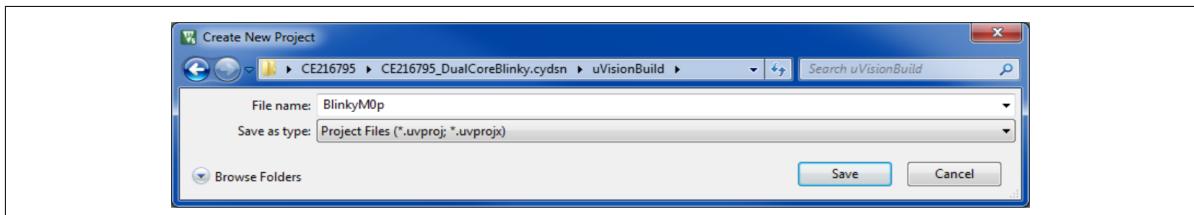


Figure 417 Create a µVision project for CM0+

5 PSoC™ 6 application notes

~~DRAFT~~

After you click **Save**, a **Select Device for Target 'Target 1'...** dialog box is displayed. The two PSoC™ 6 MCU CPUs that were defined in the previously installed pack (Figure 415) are displayed. Select the CM0+ CPU, as Figure 418 shows. Click **OK**.

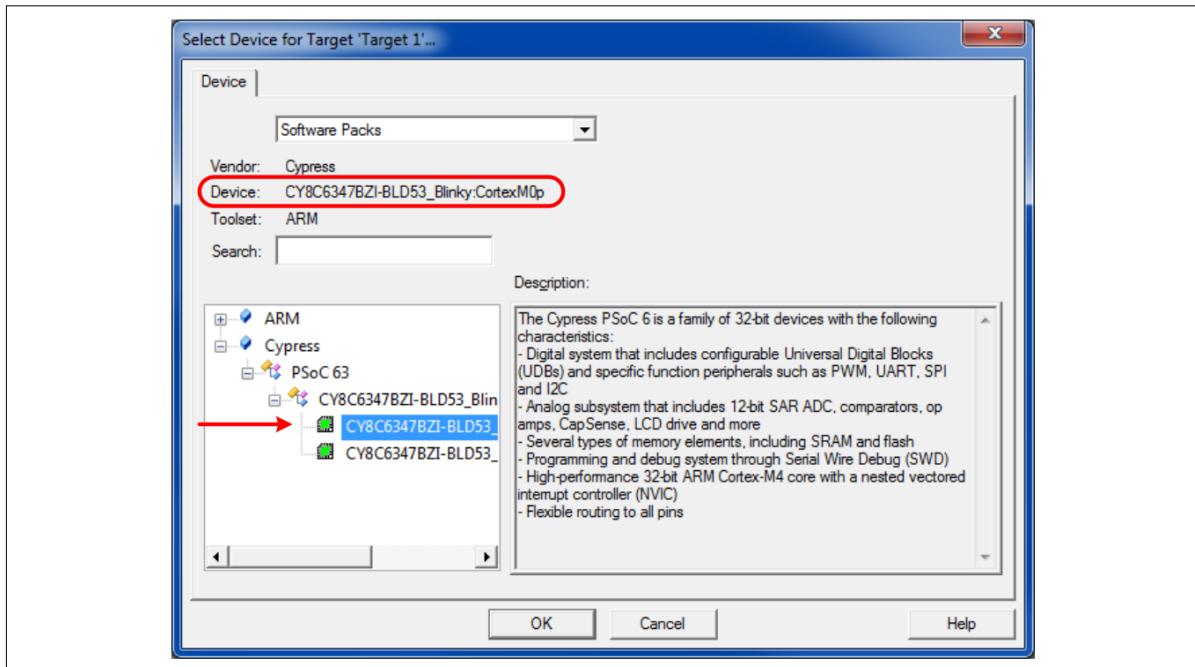


Figure 418 Select CM0+ as the project device

Next, a **Manage Run-Time Environment** dialog box is displayed. Click **Select Packs** and uncheck **Use latest versions of all installed Software Packs**. Select the pack from the PSoC™ Creator project, as Figure 419 shows:

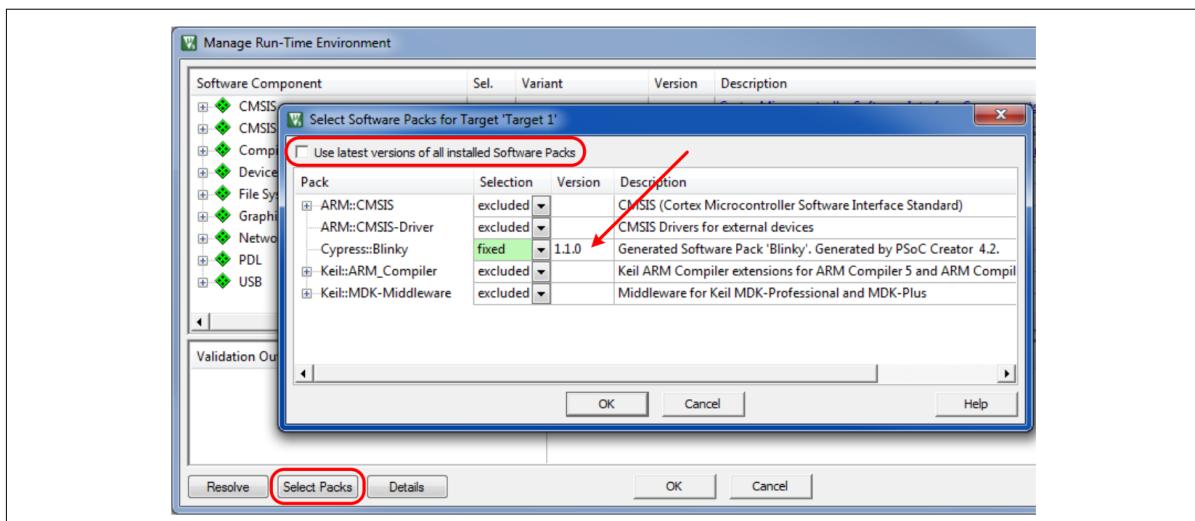


Figure 419 Select the PSoC™ Creator project pack

Click **OK**; the **Manage Run-Time Environment** dialog changes as Figure 420 shows. Select the Device Startup and PDL Drivers and click **OK**. The project is created, with a Target 1, a Source Group 1, and Device startup and PDL files, as Figure 421 shows

5 PSoC™ 6 application notes

DRAFT

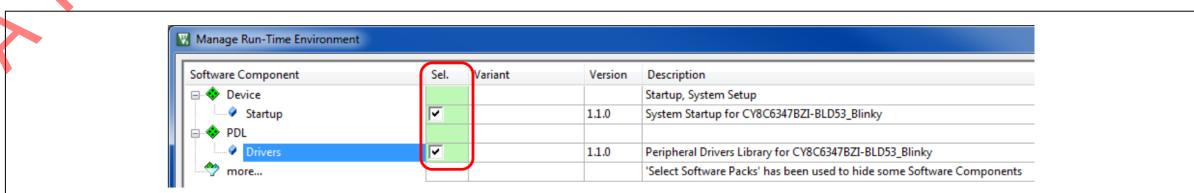


Figure 420 Select pack start-up and PDL driver files

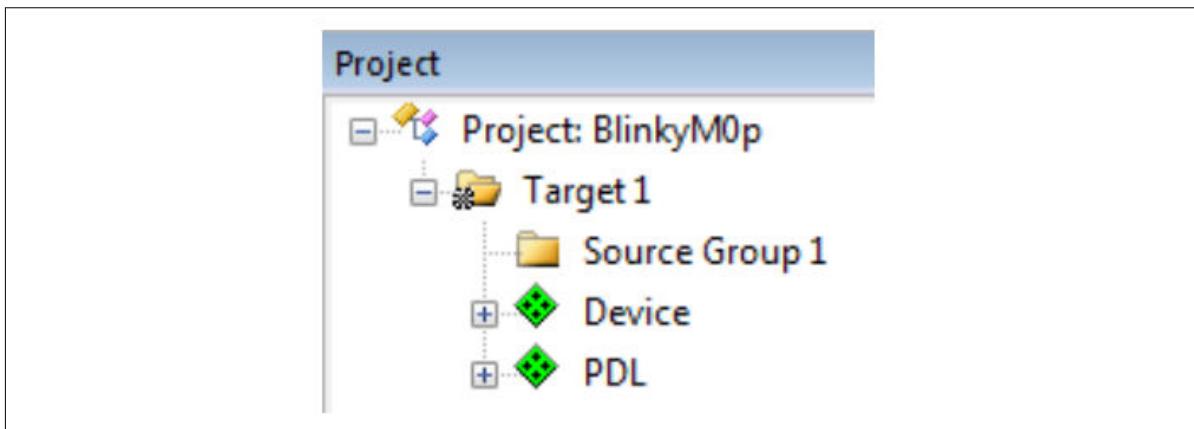


Figure 421 Initial project creation

Right-click **Source Group 1**, and select **Add Existing Files to Group ‘Source Group 1’...**. Navigate to your PSoC™ Creator project folder and select `main_cm0p.c`, `cy_ipc_config.c`, and all other non-system .c and assembler files needed for your project, as Figure 422 shows. You do not have to add any .h files, startup, or system .c, or assembler files. Click **Add**; the files are added to the source group in the µVision project. Click **Close**.

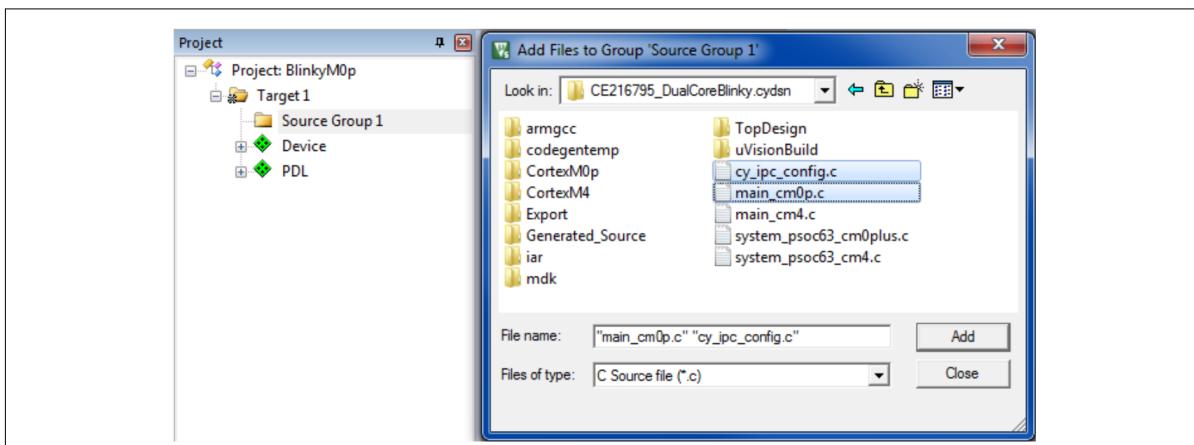


Figure 422 Add PSoC™ Creator project C source files to the source group

Now that the project is created, you must set its options. Right-click **Target 1**, and select **Options for Target ‘Target 1’...**. Confirm in the **Target** tab that the device, CPU, IROM1, and IRAM1 are correct for your PSoC™ 6 MCU device, as Figure 423 shows. Updating other fields such as Xtal (MHz) and Operating system is optional

5 PSoC™ 6 application notes

DRAFT

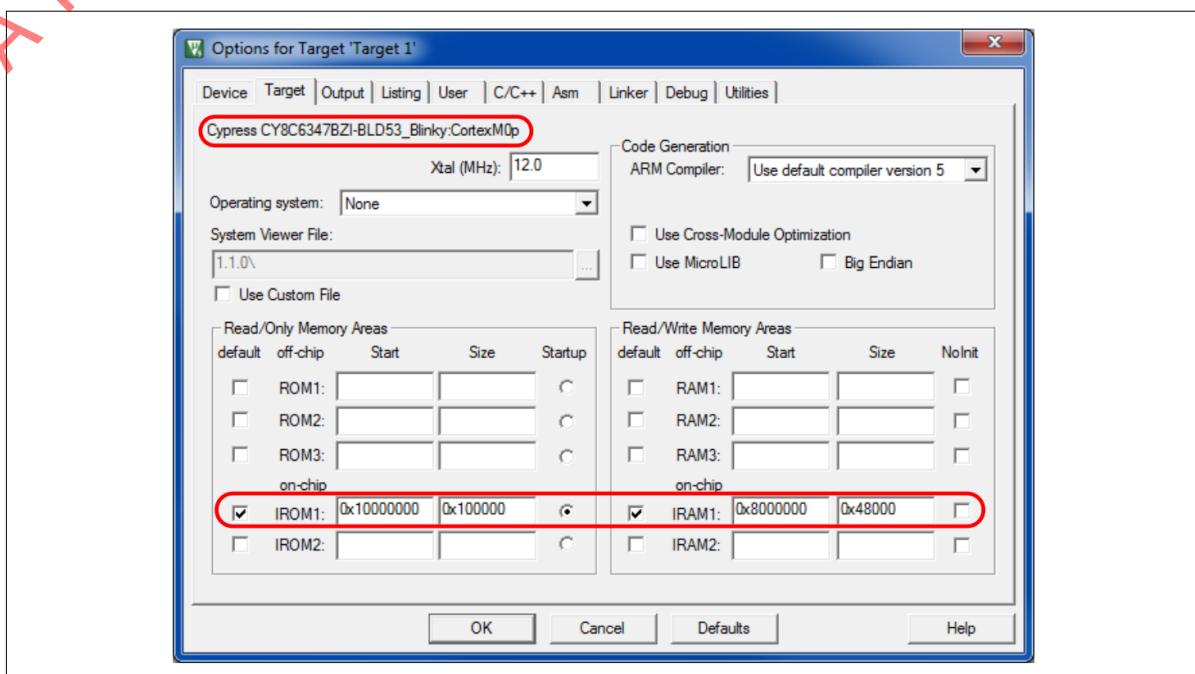


Figure 423 Project target options

In the **User** tab, verify that the correct post-build batch file from the pack is being called. Hover the cursor over the **User Command** field and confirm that `postbuildCortexM0p.bat` is called, as Figure 424 shows. Add other pre- and post-build batch files, and select other options, as needed

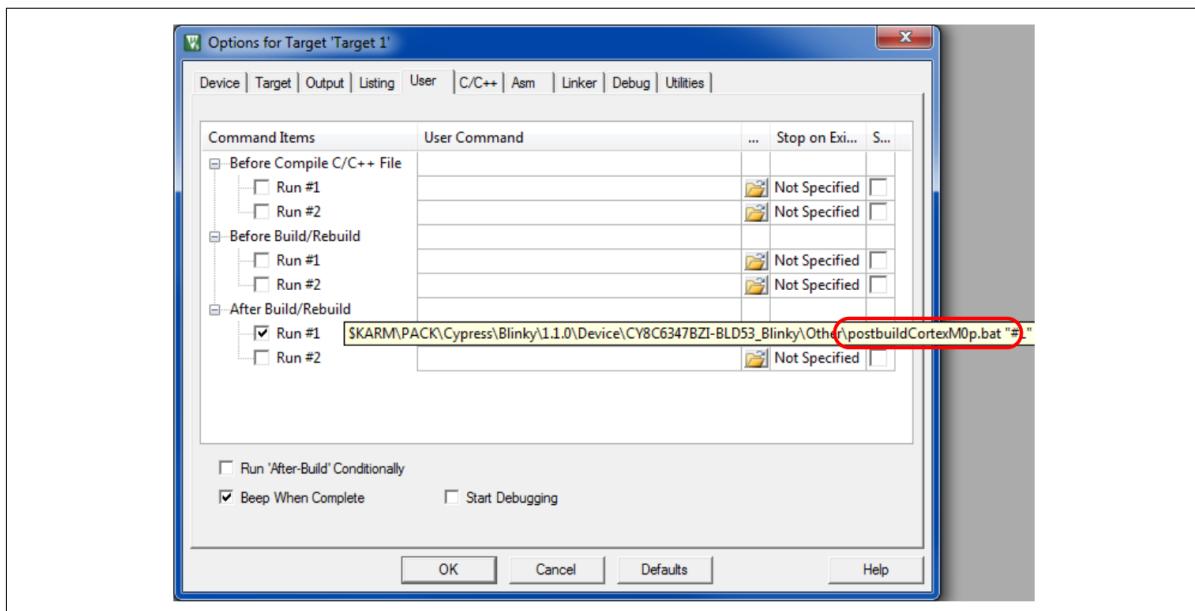


Figure 424 Project user options

Confirm in the **C/C++** tab that the **C99 mode** option is checked, as Figure 425 shows. (PDL is developed based on C99.) Add the PSoC™ Creator <project>.cydsn folder to the **Include Paths**; this provides a link to the .h files in the PSoC™ Creator project. Update other options and fields as needed

5 PSoC™ 6 application notes

DRAFT

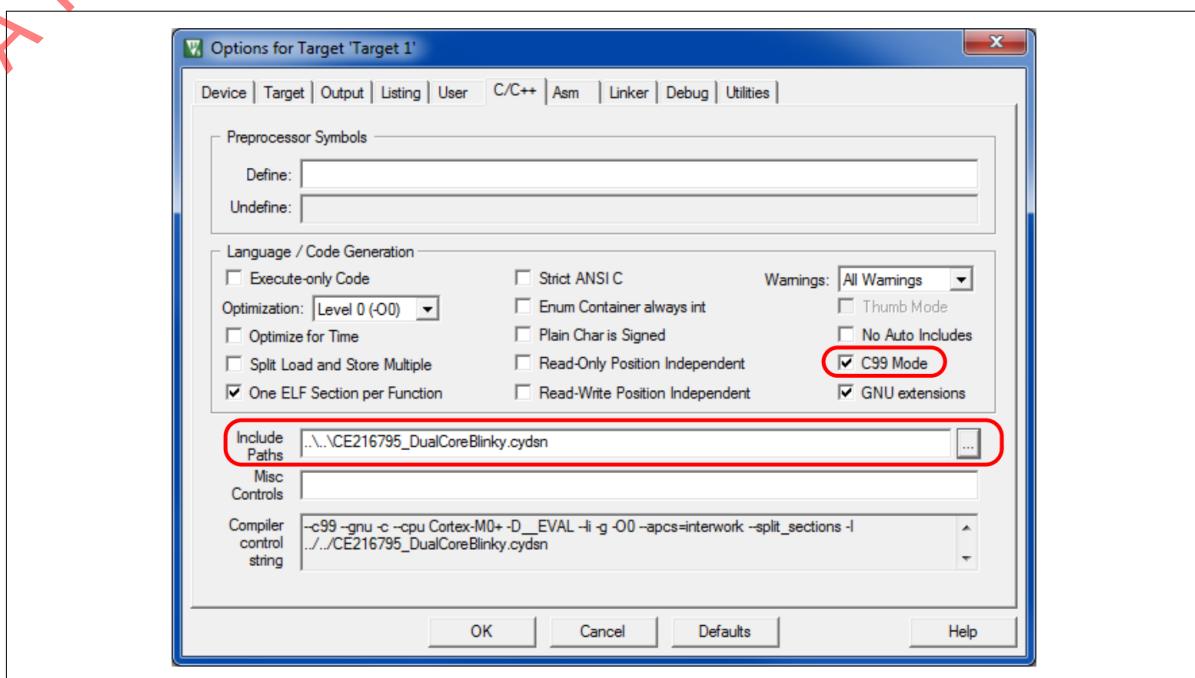


Figure 425 Project C/C++ options

Confirm in the **Linker** tab that the **R/O Base** and **R/W Base** fields are correct for your PSoC™ 6 MCU device, as [Figure 426](#) shows. Select the appropriate **Scatter File** from your PSoC™ Creator project folder

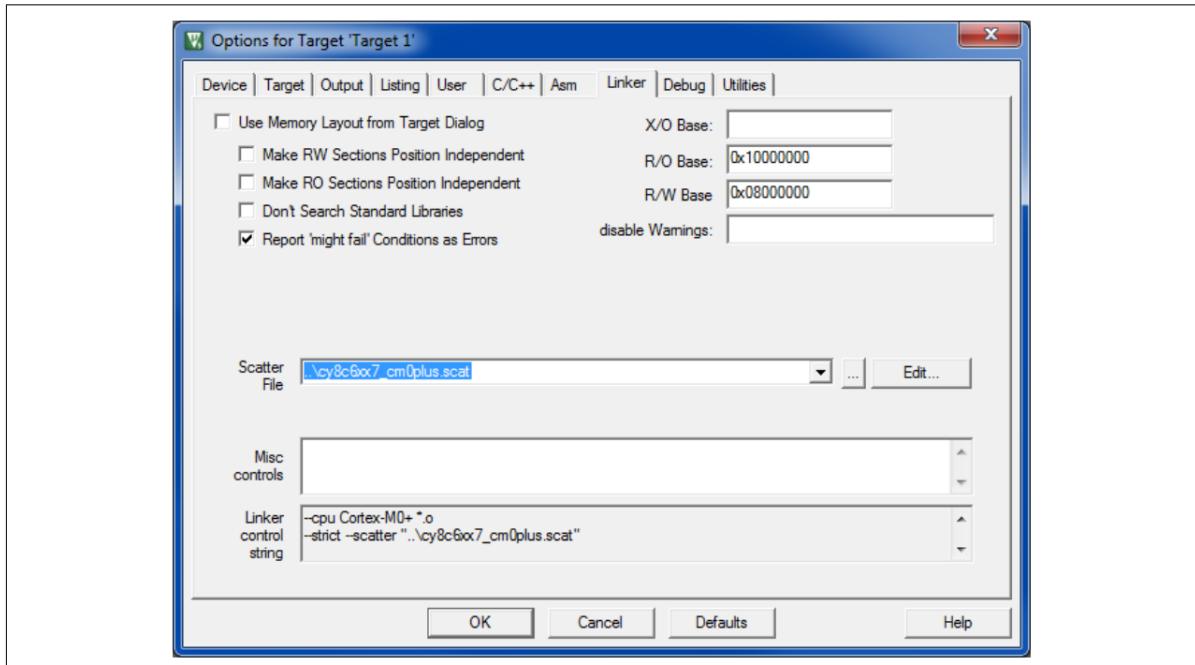


Figure 426 Project linker options

Connect the CY8CKIT-062-BLE USB port to your computer. Press the kit button SW3 to put KitProg2 into CMSIS-DAP mode; see the kit guide for details. This allows debugging without using any external probes

In the **Debug** tab, select **Use CMSIS-DAP Debugger**, as [Figure 427](#) shows. Click **Settings**, select **KitProg2 CMSIS-DAP**, and confirm that all other settings are at the defaults shown. Click **OK** and go back to the Options dialog

5 PSoC™ 6 application notes

DRAFT

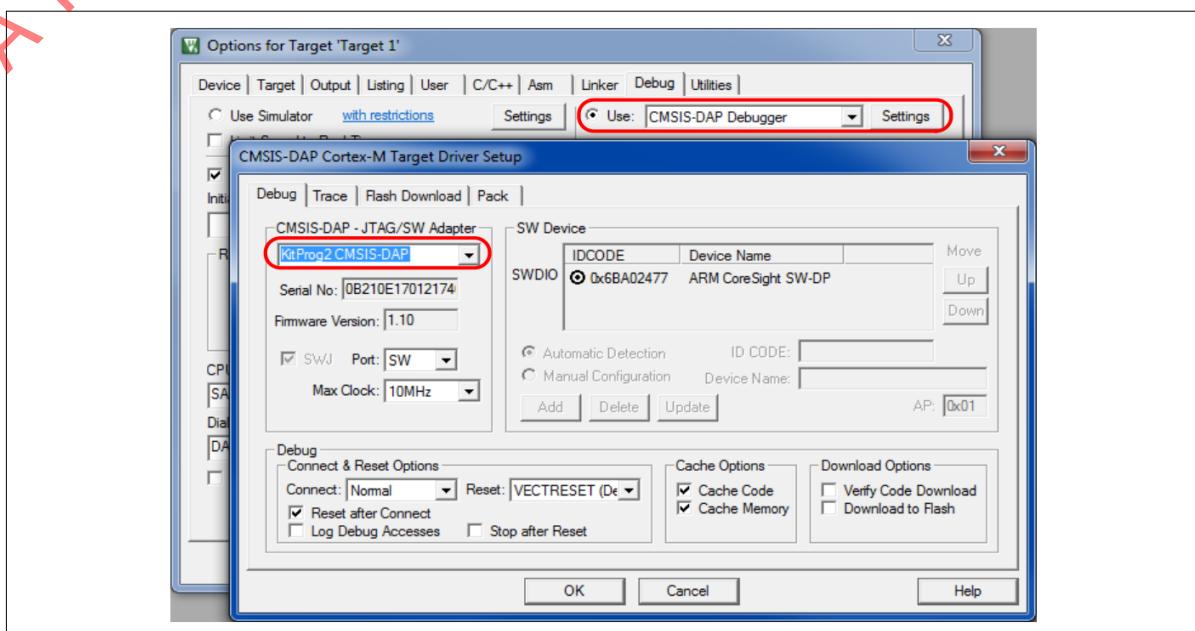


Figure 427 Project debug options

In the **Utilities** tab, confirm that Use Debug Driver is checked, and then uncheck **Update Target before Debugging**. Click **Settings**, and uncheck all **Download Function** boxes, as Figure 428 shows. Click **Do not Erase**. Click **OK** and go back to the **Options** dialog. A warning “Nothing to do ...” is displayed; click **OK**. The application will be loaded by the CM4 project. Click **OK** to save and close the options settings

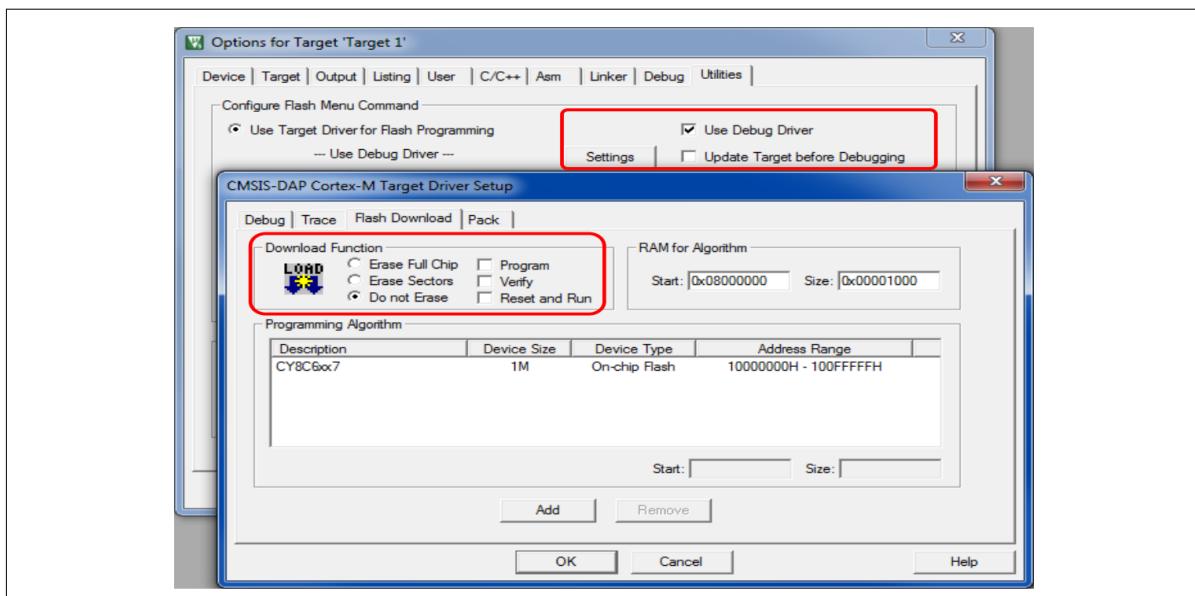


Figure 428 Project utilities options

Repeat the previous steps and create a second project for CM4.

Recommended: name the project based on the original PSoC™ Creator project name and the target CPU. For example, for the [CE216795](#) dual-core blinky project, create μVision project **BlinkyM4p**; see [Figure 417](#). Configure the project in the same manner as the CM0+ project, with the following differences:

- The CM4 project must be in the same folder as the CM0+ project; in this case, **μVisionBuild**. See [Figure 417](#)
- Select the CM4 CPU from the previously installed pack; see [Figure 418](#)

5 PSoC™ 6 application notes

- Navigate to your PSoC™ Creator project folder and select `main_cm4.c`, `cy_ipc_config.c`, and all other non-system .c and assembler files needed for your project, as [Figure 422](#) shows. You do not have to add any .h files, startup, system .c, or assembler files
- In the **Options** dialog, **Output** tab, click **Select Folder for Objects...**, and select the `ObjectsM4` folder that you created; see [Figure 416](#)
- In the **Options** dialog, **C/C++** tab, add `--fpu=fpv4-sp` to **Misc Controls**; see [Figure 425](#)
- In the **Options** dialog, **Linker** tab, select the “cm4_dual” scatter file, as [Figure 429](#) shows. The CM4 project will contain code for both CPUs. Add `--fpu=fpv4-sp` to **Misc Controls**

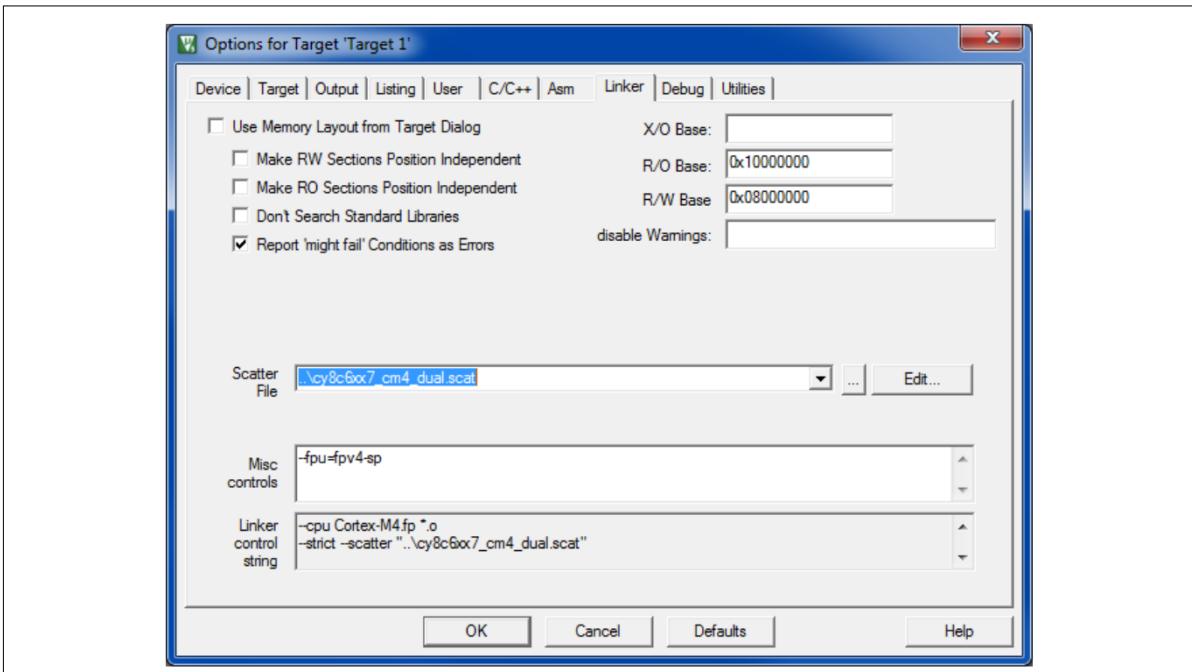


Figure 429 Linker options for CM4 project

- In the **Options** dialog, **Debug** tab, Target Driver Setup, select VECTRESET for the **Reset** option; see [Figure 427](#).
- In the **Options** dialog, **Utilities** tab, confirm that **Update Target before Debugging** is checked, as [Figure 430](#) shows. Set the RAM for Algorithm values as indicated. Checking **Reset and Run** is optional but convenient

5 PSoC™ 6 application notes

DRAFT

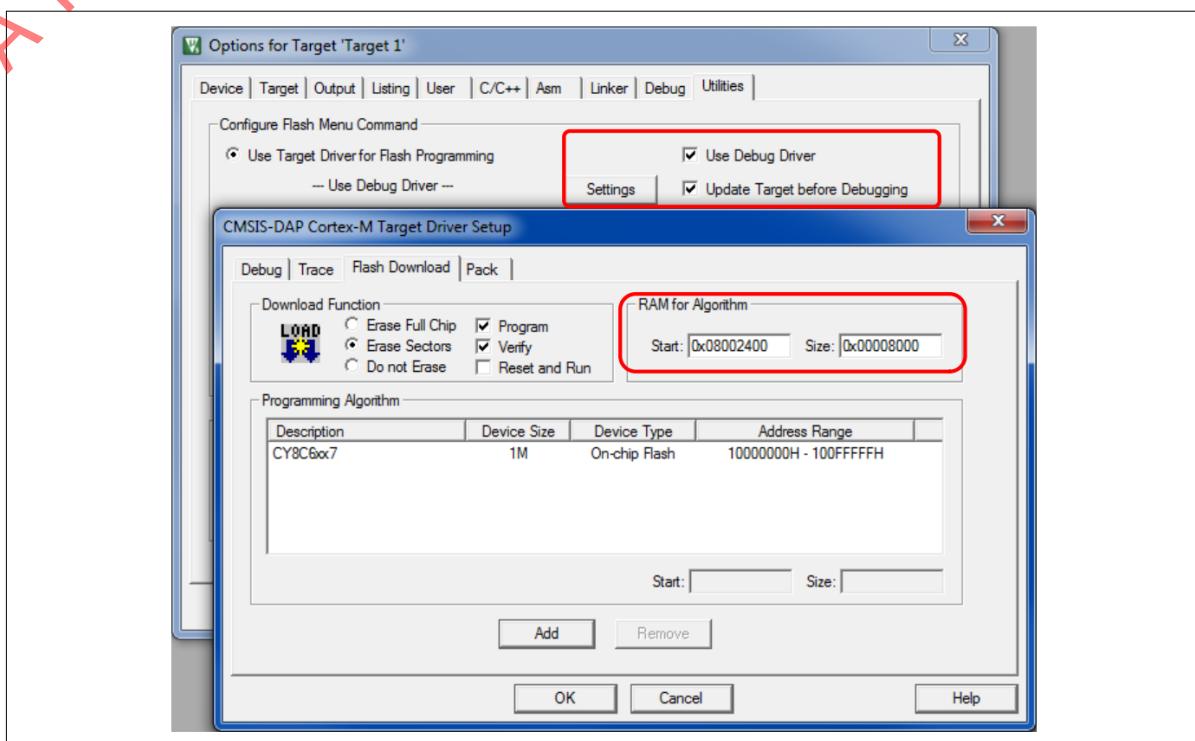


Figure 430 Utilities options for CM4 project

Finally, create a µVision workspace (**Project > New Multi-Project Workspace...**), named for example Blinky, in the uVisionBuild folder. Add the two created projects to that workspace. The created workspace and projects, and the corresponding files, should be similar to [Figure 431](#).

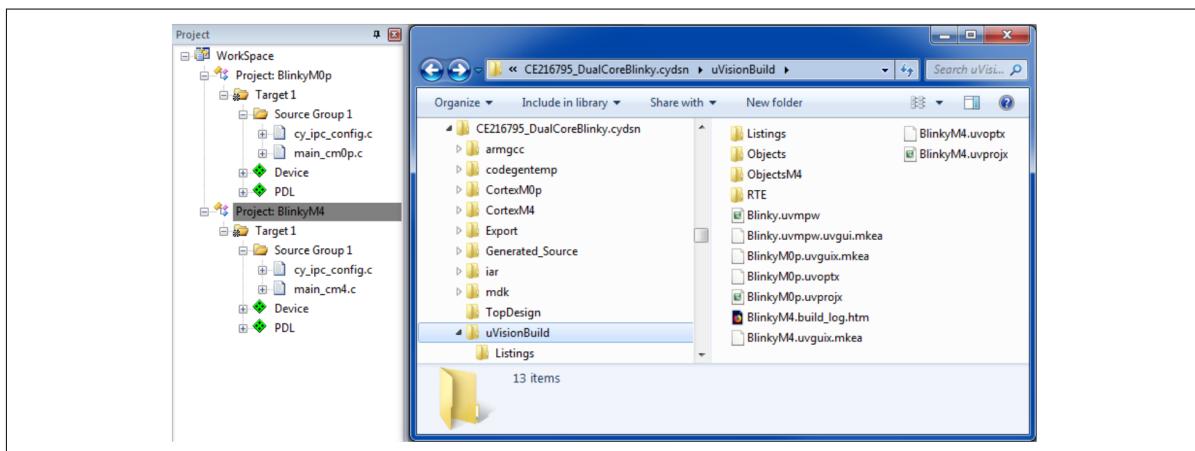


Figure 431 Resultant µVision project window and project files

Build the projects in sequence; build the CM0+ project first. Note that µVision has a batch build feature to automate the process. After building is successfully completed, right-click the BlinkyM4 project and set it as the active project. Then test your build options by (1) erasing flash (**Flash > Erase**), and (2) downloading the project (**Flash > Download**) and confirming correct operation. If you did not select Reset and Run (see [Figure 430](#)), you must press the kit reset button (RST / SW1) to start operation.

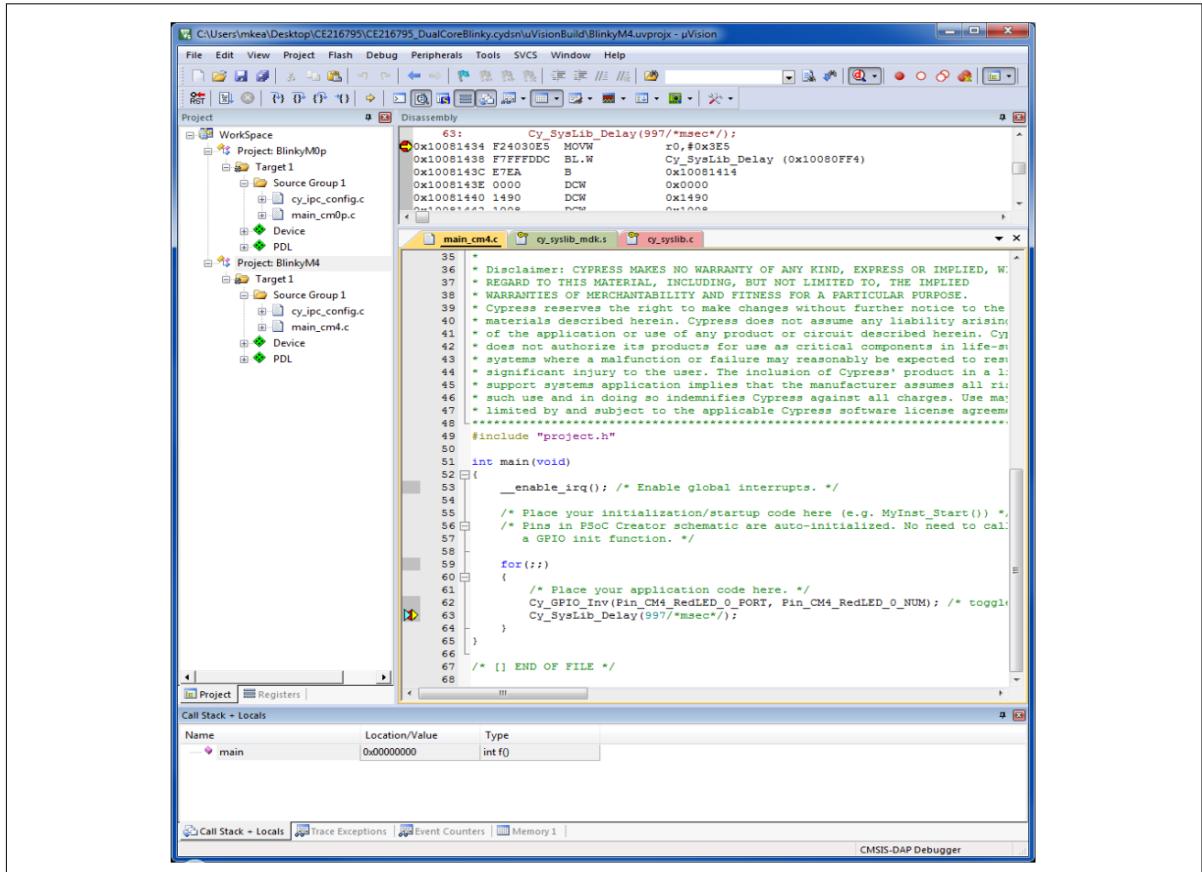
Note: *If you change any code in the CM0+ project, you must rebuild both projects. Note that µVision has a batch build feature to automate the process.*

3. Debug µVision projects

5 PSoC™ 6 application notes

~~DRAFT~~

Start debugging with the CM4 project – downloading the CM4 project installs code for both CPUs. Set the CM4 project as the active project, download it if needed, and click **Debug > Start/Stop Debug Session** to start debugging. The µVision window appears similar to [Figure 432](#).

**Figure 432****CM4 debug window**

If you are running the [CE216795](#) dual-core blinky project, set a breakpoint at line 63, `Cy_Syslib_Delay()`. Then repeatedly click **Debug > Run**, and the red LED toggles on each stop at the breakpoint

Now open a second instance of µVision and load the same workspace. Both instances share the kit connection and the PSoC™ 6 MCU debug access port (DAP). Make the CM0+ project active, and start a debug session. Set a breakpoint at line 63, `Cy_Syslib_Delay()`. Then repeatedly click **Debug > Run**, and the blue LED toggles on each stop at the breakpoint

Note: Executing the `cy_SysEnabLecM4()` function call at line 55 causes CM4 to start running again. Go to the CM4 window, click **Debug > Stop**, then **Debug > Run**. CM4 runs to the breakpoint again.

It helps to place the instance windows side by side on your desktop. The windows appear similar to [Figure 433](#). Click in the appropriate window to perform a debug operation on the desired CPU. Note that breakpoints can be set separately for each CPU. You can read and update the same memory addresses from either window

5 PSoC™ 6 application notes

DRAFT

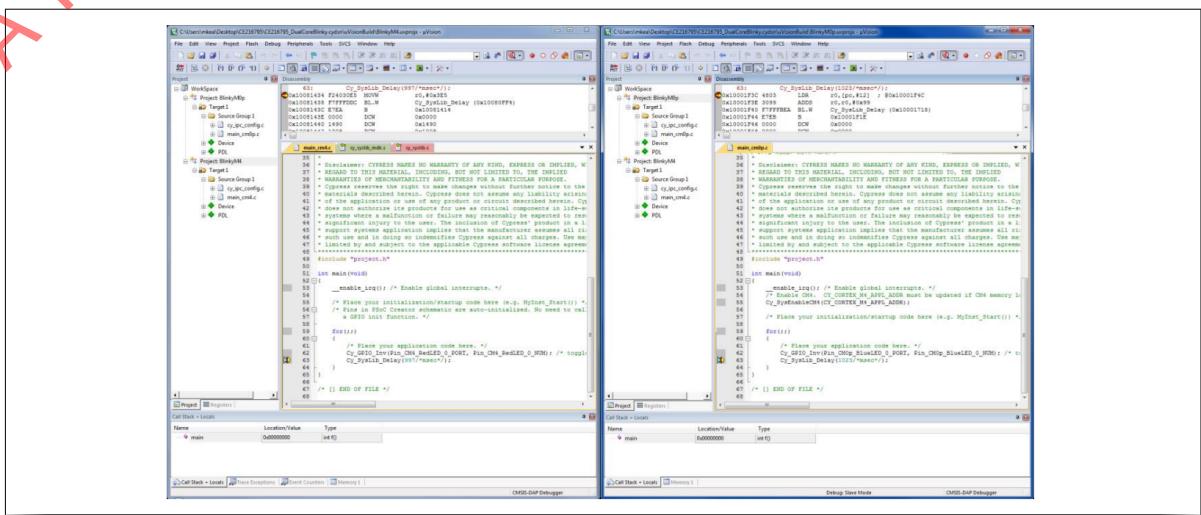


Figure 433 **µVision dual-core debugging**

4. Create IAR-EW projects

For IAR Embedded Workbench (IAR-EW), you must create two projects: one for each PSoC™ 6 MCU CPU: CM0+ and CM4. Do the following:

Note: *The IAR-EW project files should be created in your PSoC™ Creator <project>.cydsn folder. Do not create a separate folder within your PSoC™ Creator <project>.cydsn folder (this is different from the µVision instructions). Recommended: add a tag such as “IAR_” to each project and workspace file name, to distinguish the IAR-EW files from the PSoC™ Creator files in the same folder.*

Open IAR Embedded Workbench for ARM® 8.22 or later and create a new project (**Project > Create New Project...**). In the Create New Project dialog (Figure 434), confirm that the Tool chain is **Arm®**, select the **Empty project** template, then click **OK**.

5 PSoC™ 6 application notes

DRAFT

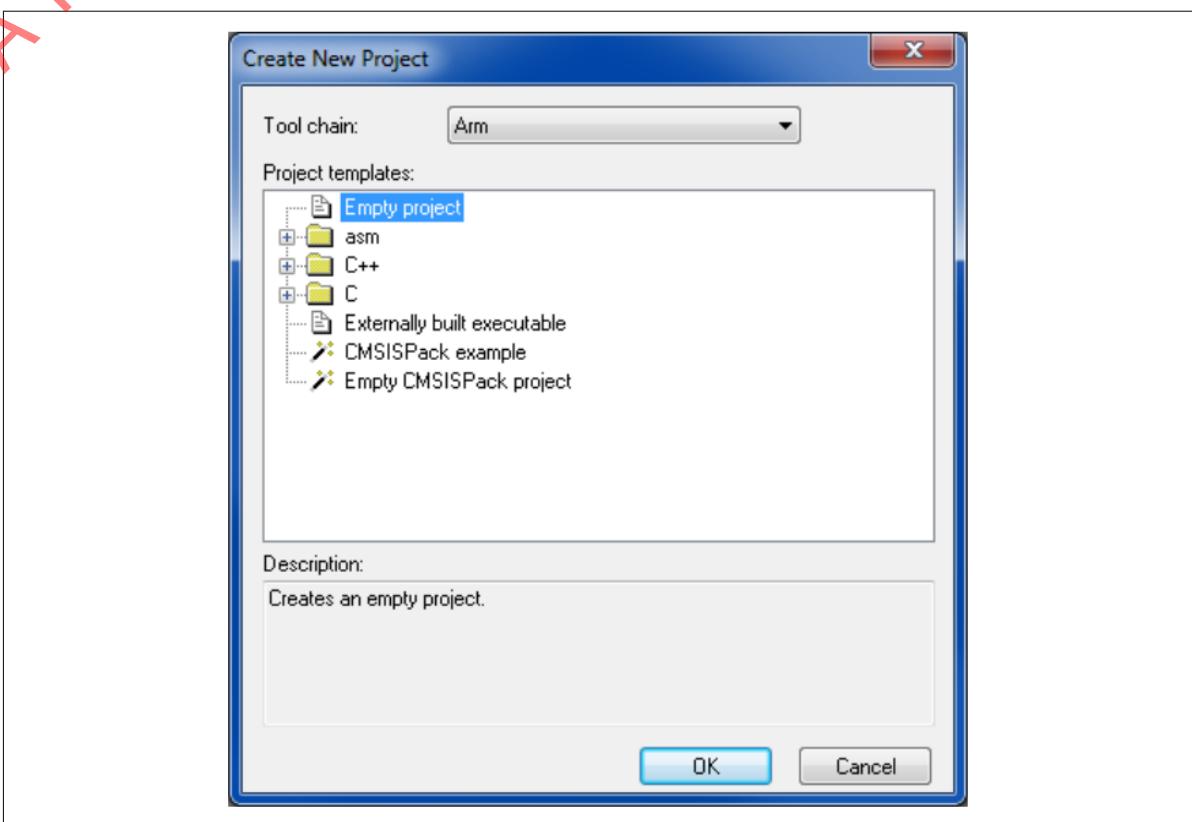


Figure 434 IAR Embedded Workbench create new project dialog

Recommended: in the Save As dialog (Figure 435), name the project based on the original PSoC™ Creator project name and the target CPU. For example, for the [CE216795](#) dual-core blinky project, create a µVision project IAR_BlinkyM0p for the CM0+ CPU

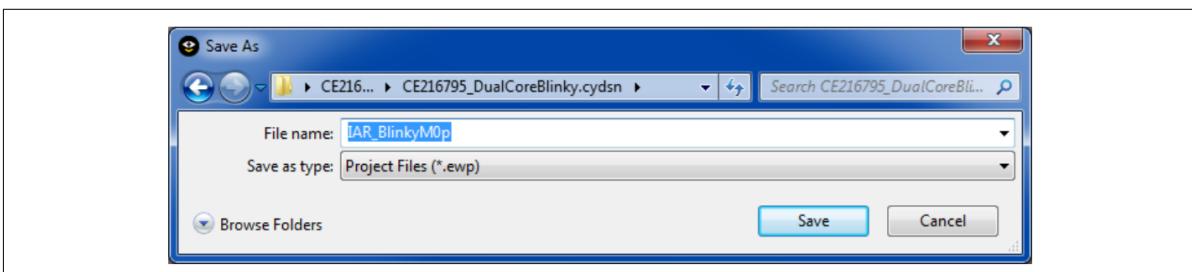


Figure 435 Create an IAR Embedded Workbench project for CM0+

Select **Tools > Options** and make sure that **Enable project connections** is checked. Click **OK**. Then select **Project > Add Project Connection....** In the next dialog, select Connect using **IAR Project Connection**, and click **OK**. Then select the ...CortexM0p.ipcf file, as [Figure 436](#) shows. Click **OK**, and several folders and files are added to the project in the Workspace window

5 PSoC™ 6 application notes

DRAFT

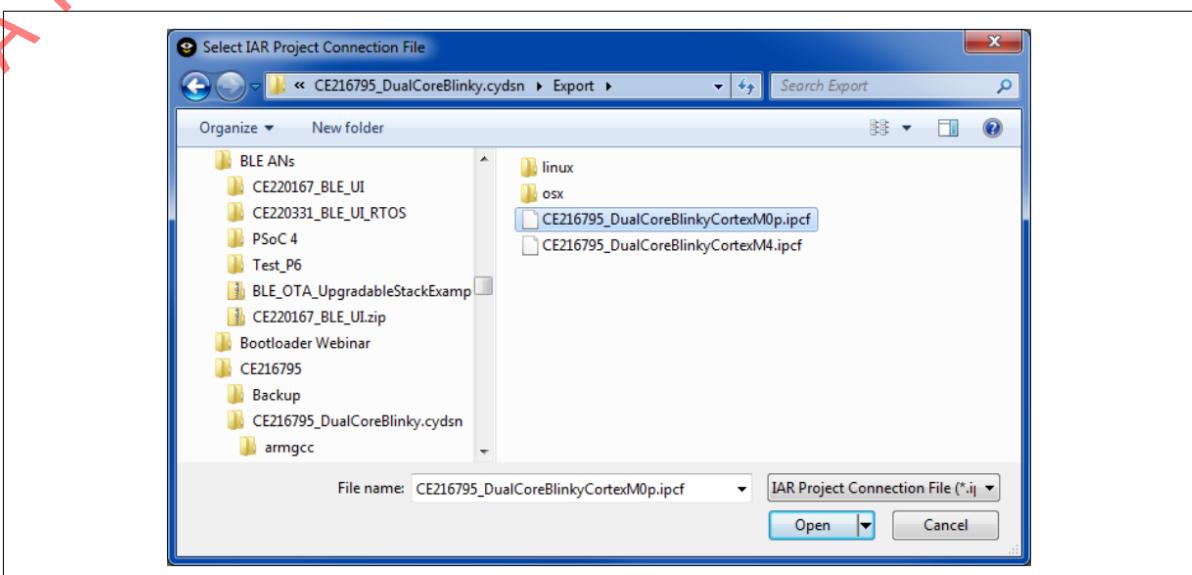


Figure 436 Select IAR project connection file from PSoC™ Creator project export folder

Now that the project is created, you must set its options. Right-click the project, and select **Options....**. Confirm in the Options dialog, **Build Actions** section that postbuildCortexM0p.bat is called, as Figure 437 shows

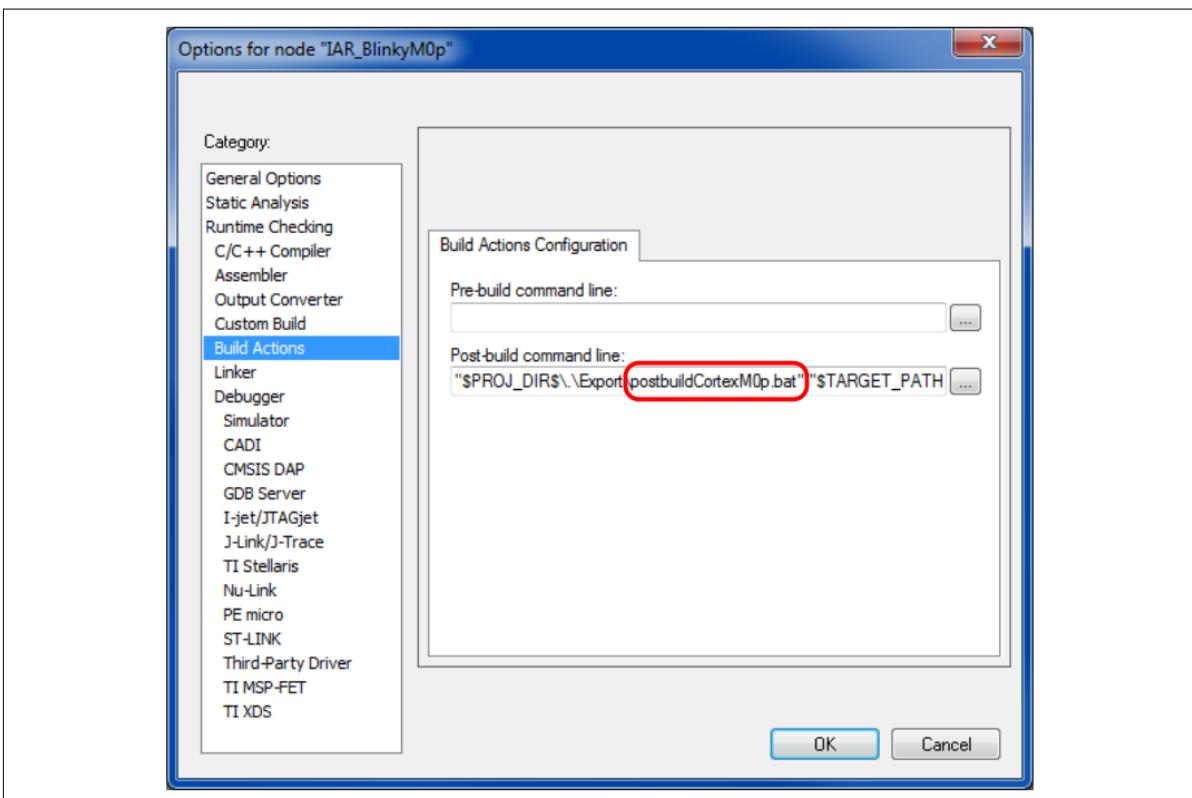


Figure 437 Select PSoC™ Creator post-build batch file

In the **Debugger** section, **Setup** tab, select the **CMSIS DAP** driver. In the **Download** tab, check **Suppress download**. In the **CMSIS DAP** section, **Setup** tab, set **Reset to Disabled (no reset)**. The application will be loaded by the CM4 project. In the **Interface** tab, select **SWD**. Click **OK**.

5 PSoC™ 6 application notes

~~DRAFT~~

Repeat the previous steps and create a second project for CM4. **Recommended:** name the project based on the original PSoC™ Creator project name and the target CPU. For example, for the CE216795 dual-core blinky project, create IAR-EW project IAR_BlinkyM4; see [Figure 435](#). Configure the project similar to the CM0+ project, with the following differences:

- The CM4 project must be in the same folder as the CM0+ project; in this case, your PSoC™ Creator <project>.cydsn folder. See [Figure 435](#)
- Select the ...CortexM4.ipcf file; see [Figure 436](#)
- In the **Options** dialog, **General Options** section, **Output** tab, change the output directories for object and list files, as [Figure 438](#) shows. Do not change the executables/libraries output folder

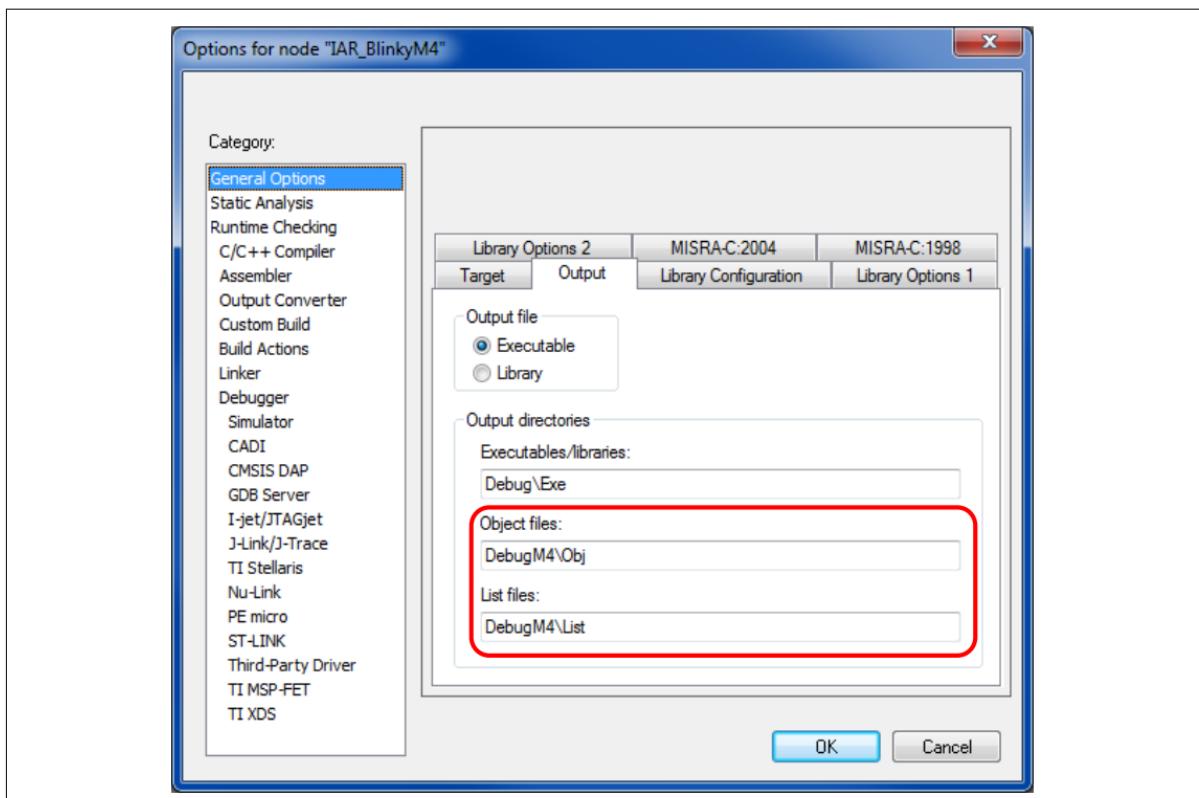


Figure 438 Unique output folders for CM4 project

- In the **Build Actions** section, confirm that postbuildCortexM4.bat is called, see [Figure 437](#)
- In the **Debugger** section, **Setup** tab, select the **CMSIS DAP** driver. In the **CMSIS DAP** section, **Setup** tab, confirm that **Reset** is set to **System (default)**. In the **Interface** tab, select **SWD**. Click **OK**

Save Workspace As dialog, create an IAR-EW workspace, named for example IAR_Blinky, in your PSoC™ Creator <project>.cydsn folder. The created workspace and projects, and the corresponding files, should be similar to [Figure 439](#). The files and folders generated by IAR-EW are highlighted

5 PSoC™ 6 application notes

DRAFT

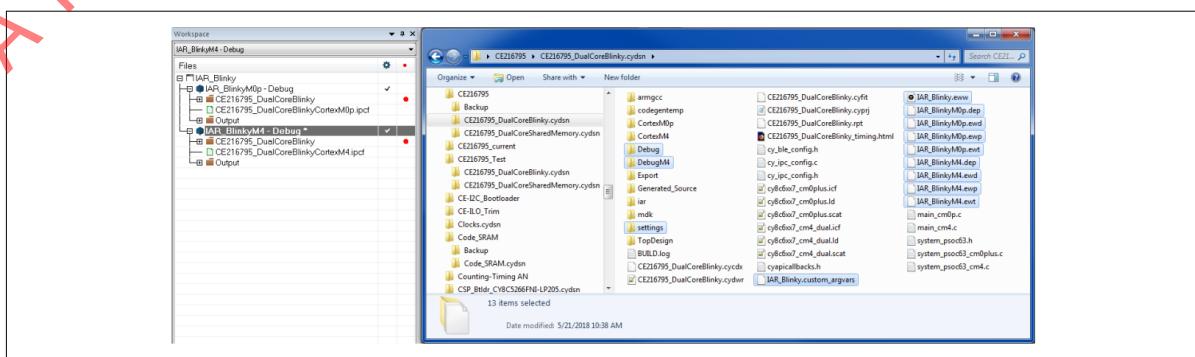


Figure 439 Resultant IAR Embedded Workbench project window and project files

Connect the CY8CKIT-062-BLE USB port to your computer. Press kit button SW3 to put KitProg2 into CMSIS-DAP mode; see the kit guide for details. This allows debugging without using any external debug probes

Build the projects in sequence; build the CM0+ project first. Note that IAR-EW has a batch build feature to automate the process. After building is successfully completed, right-click the BlinkyM4 project and set it as the active project. Then confirm that your build options are correct, by (1) erasing flash (**Project > Download > Erase memory**), and (2) downloading the project (**Project > Download > Download active application**) and confirming correct operation. After downloading, press the kit reset button (RST / SW1) to start operation

Note: When erasing flash, you typically only need to erase PSoC™ 6 MCU application flash (0x1000 0000 – 0x100F FFFF), as [Figure 440](#) shows.

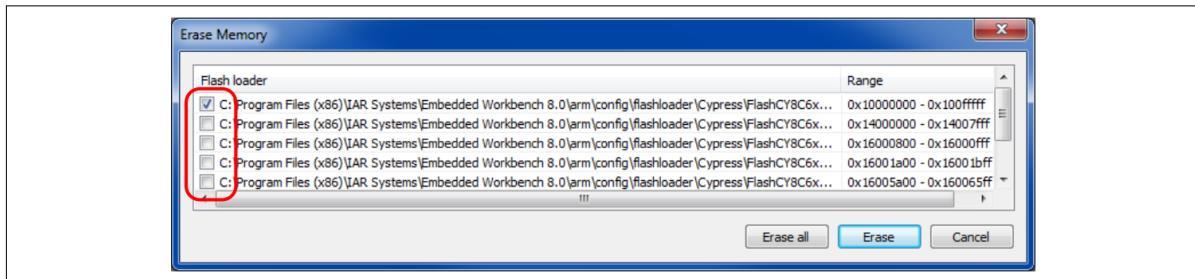


Figure 440 IAR Embedded Workbench erase memory dialog for PSoC™ 6 MCU

Note: If you change any code in the CM0+ project, you must rebuild both projects. Note that IAR-EW has a batch build feature to automate the process.

5. Debug IAR-EW projects

Reopen the options for the CM4 project, and go to the **Debugger** section, **Multicore** folder. PSoC™ 6 MCU has different cores, for example, CM0+ and CM4, which is referred to as "asymmetric multicore". Therefore, fill in the fields in the **Asymmetric multicore** section as [Figure 441](#) shows. Checking **Enable multicore master mode** makes the CM4 CPU the master for download and debugging purposes. Do not change the **Port**.

5 PSoC™ 6 application notes

DRAFT

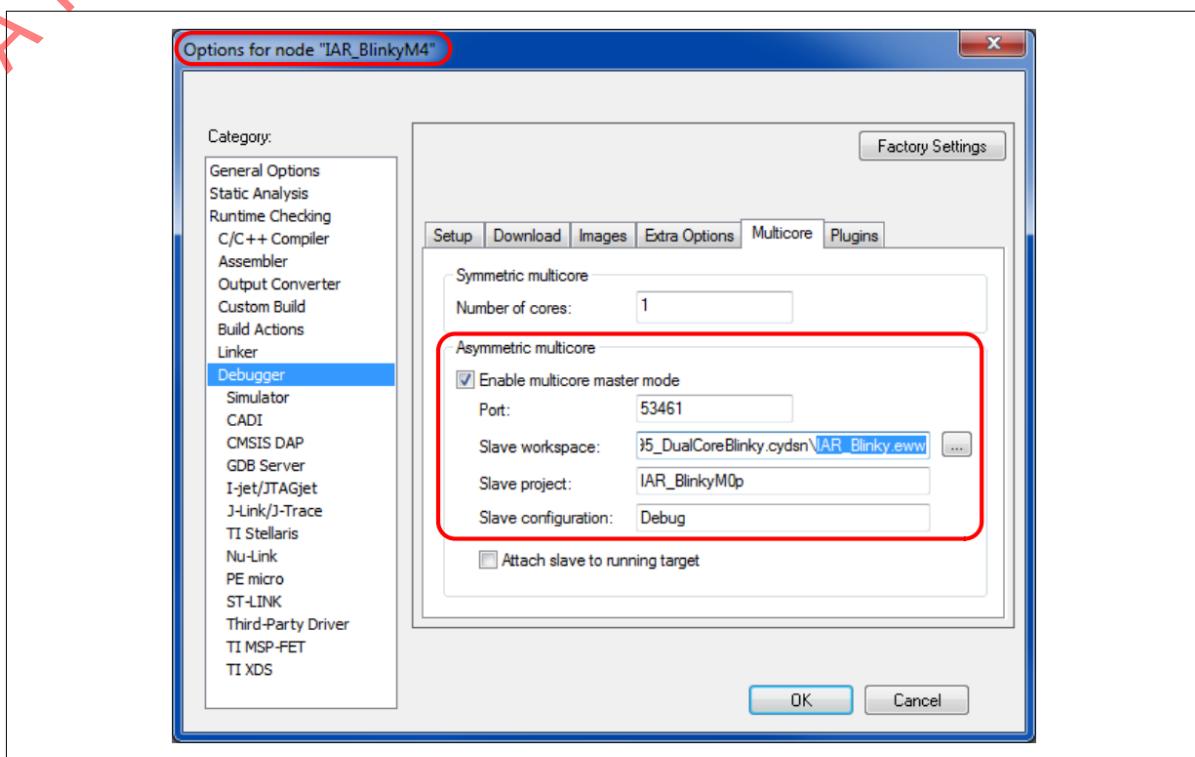


Figure 441 Set up multicore debugging

Select **File > Save All** to save the project options changes. Then start debugging by selecting either **Project > Download and Debug** or **Project > Debug without Downloading**. A second (slave) instance of IAR Embedded Workbench is automatically opened for the CM0+ project. Both instances share the kit connection and the PSoC™ 6 MCU debug access port (DAP).

In the slave instance, set a breakpoint at line 63, `Cy_Syslib_Delay()`. Then repeatedly click **Debug > Go**, and the blue LED toggles on each stop at the breakpoint

Click anywhere in the CM4 instance window and repeat the process. The red LED toggles on each stop at the breakpoint

It helps to place the instance windows side by side on your desktop. The windows appear like [Figure 442](#). Click in the appropriate window to perform a debug operation on the desired CPU. Note that breakpoints can be set separately for each CPU. You can read and update the same memory addresses from either window

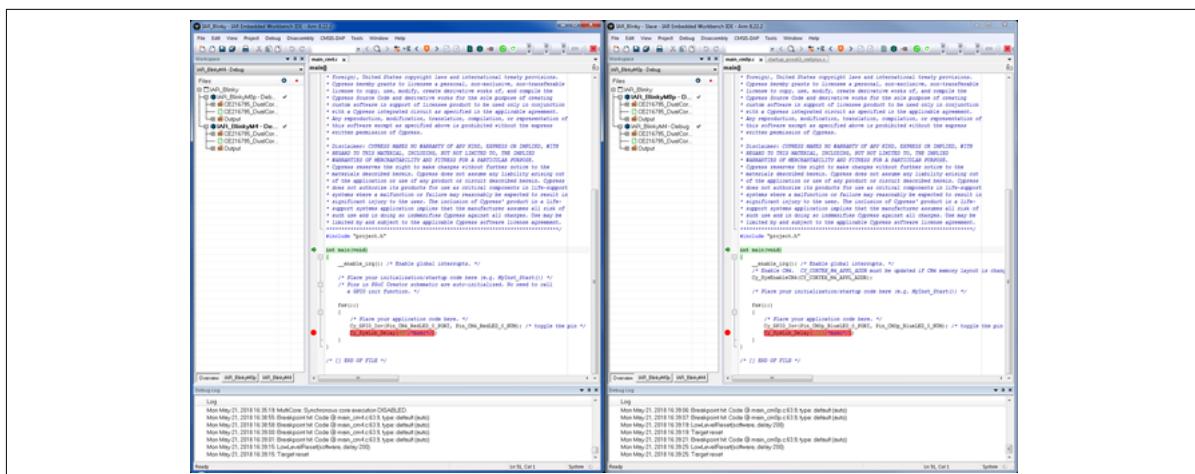


Figure 442 IAR Embedded Workbench dual-CPU debugging

5 PSoC™ 6 application notes

You can stop debugging in either window; debugging is ended for both CPUs. Press the kit reset button (RST/SW1) to restart kit operation

DRAFT

5 PSoC™ 6 application notes

5.10.5 ~~DRAFT~~ Summary

This application note has shown how to use and optimize your firmware and hardware designs for the dual-core feature in PSoC™ 6 MCUs.

Another way to optimize your PSoC™ 6 MCU design is based on the fact that PSoC™ devices are designed to be flexible and enable you to build custom functions in programmable analog and digital blocks. For example, PSoC™ 6 MCU has the following peripherals that can act as “co-processors”:

- **DMA controllers** Note that the most common CPU assembler instructions output by C compilers are MOV, LDR, and STR, which implies that the CPU spends a lot of cycles just moving bytes around. Let the DMA controllers do that instead

Note: PSoC™ 6 MCU DMA controllers have an extensive set of features that enable you to construct complex data transfer and control systems that are independent of the CPUs. Software support of these features is provided in the ModusToolbox™ software Device Configurator, a PSoC™ Creator DMA Component, and an API in the PDL. For more information, see the Device Configurator Help Guide, the DMA Component datasheet, or the PDL documentation.

- **Crypto block** This block offers hardware acceleration for symmetric and asymmetric cryptographic methods (AES, 3DES, RSA, and ECC) and hash functions (SHA-512, SHA-256). It also has a true random number generator (TRNG) function. Software support for these features is provided by an API in the PDL; see the PDL documentation. The Infineon HAL does not have a crypto driver at the moment
- **Universal Digital Blocks (UDBs)** There are as many as 12 UDBs, and each UDB has an 8-bit datapath that can add, subtract, and do bitwise operations, shifts, and cyclic redundancy check (CRC). Datapaths can be chained for word-wide calculations. Consider offloading CPU calculations to the datapaths. At the moment, only PSoC™ Creator supports UDBs. ModusToolbox™ software does not have any support for UDBs
- UDBs also have programmable logic devices (PLDs) which can be used to build state machines; see for example the Lookup Table (LUT) Component datasheet. LUTs can be an effective hardware-based alternative to programming state machines in the CPU, for example by using C switch/case statements.
In addition, two GPIO ports include Smart I/O™, which can be used to perform Boolean operations directly on signals going to, and coming from, GPIO pins. The Infineon HAL does not have support for Smart I/O
- Other smart peripherals include serial communication blocks (SCB), counter/timer/PWM blocks (TCPWM), Bluetooth® Low Energy (BLE), I2S/PDM audio, programmable analog, and CAPSENSE™. Use these peripherals to further offload processing from the CPUs

PSoC™ Creator and ModusToolbox™ software offer many Components and extensive APIs in the PDL to support the peripherals’ functions. This allows you to develop an effective multiprocessing system in a single chip, offloading many functionalities from the CPUs. This, in turn, can not only reduce code size, but by reducing the number of tasks that the CPUs must perform, presents an opportunity to reduce CPU speed and power consumption.

For example, you can implement a digital system to control multiplexed ADC inputs, and interface with DMA to save the data in the SRAM to create an advanced analog data collection system with zero usage of the CPUs.

ModusToolbox™ software provides a set of tools for setting up peripherals, pre-defined BSPs for all Infineon kits, libraries for popular functionality like CAPSENSE™ and emWin, and a comprehensive array of example applications to get you started.

Infineon offers extensive application note and [code example](#) support for PSoC™ peripherals, as well as detailed data in the device datasheets, PDL documentation, HAL documentation, and technical reference manuals (TRMs). For more information, see Related Documents.

5 PSoC™ 6 application notes

References

-
- 1
- 0

For a comprehensive list of PSoC™ 6 MCU resources, see [KBA223067](#) in the Infineon Support.

Application notes

AN228571 - Getting started with PSoC™ 6 MCU on ModusToolbox™ software	Describes PSoC™ 6 MCU devices and how to build your first ModusToolbox™ project
AN221774 – Getting started with PSoC™ 6 MCU on PSoC™ Creator	Describes PSoC™ 6 MCU devices and how to build your first ModusToolbox™ or PSoC™ Creator project
AN210781 – Getting started with PSoC™ 6 MCU with Bluetooth® low energy connectivity on PSoC™ Creator	Describes PSoC™ 6 MCU with BLE Connectivity devices and how to build your first PSoC™ Creator project
AN217666 – PSoC™ 6 MCU interrupts	Describes PSoC™ 6 MCU interrupt architecture and how to configure interrupts
AN219434 – Importing PSoC™ Creator Code into an IDE for a PSoC™ 6 MCU Project	Describes how to import the code generated by PSoC™ Creator into your preferred IDE

Code examples (PSoC™ Creator)

CE216795 – PSoC™ 6 MCU dual-CPU basics	Demonstrates the two CPU cores in PSoC™ 6 MCU doing separate independent tasks and communicating with each other using shared memory and the inter-processor communication (IPC) block
CE223820 – PSoC™ 6 MCU IPC pipes	This example demonstrates how to use the inter-processor communication (IPC) driver to implement a message pipe in PSoC™ 6 MCU. The pipe is used to send messages between CPUs
CE223549 – PSoC™ 6 MCU IPC semaphore	This example demonstrates how to use the inter processor communication (IPC) driver to implement a semaphore in PSoC™ 6 MCU. The semaphore is used as a lock to control access to a resource shared by the CPUs
CE226306 – PSoC™ 6 MCU power measurements	This example shows how to achieve the power measurements listed in the PSoC™ 6 MCU datasheets

Code examples (ModusToolbox™)

mtb-example-psoc6-dual-cpu-empty-app	This is a minimal starter Dual-CPU application template for PSoC™ 6 MCU devices
mtb-example-psoc6-dual-cpu-ipc-sema	This example demonstrates how to use the inter-processor communication (IPC) driver to implement a semaphore in PSoC™ 6 MCU. The semaphore is used to lock to control access to a resource shared by the CPUs and synchronize the initialization instructions
mtb-example-psoc6-dual-cpu-ipc-pipes	This example demonstrates how to use the inter-processor communication (IPC) driver to implement a message pipe in PSoC™ 6 MCU. The pipe is used to send messages between CPUs

PSoC™ Creator component datasheets

Interrupt	Supports generating interrupts from hardware signals
-----------	--

Device documentation

PSoC™ 6 MCU datasheets	PSoC™ 6 technical reference manuals
--	---

Development kit documentation

~~5 PSoC™ 6 application notes~~

CY8CKIT-062-BLE	PSoC™ 6 BLE pioneer kit
CY8CKIT-062-WiFi-BT	PSoC™ 6 Wi-Fi-BT pioneer kit
CY8CKIT-062S2-43012	PSoC™ 62S2 Wi-Fi BT pioneer kit
CY8CPROTO-063-BLE	PSoC™ 6 BLE prototyping kit
CY8CPROTO-062-4343W	PSoC™ 6 Wi-Fi BT prototyping kit
CY8CPROTO-062S3-4343W	PSoC™ 62S3 Wi-Fi BT prototyping kit
CYW9P62S1-43438EVB-01	PSoC™ 62S1 Wi-Fi BT pioneer kit
CYW9P62S1-43012EVB-01	PSoC™ 62S1 Wi-Fi BT pioneer kit

Tool documentation

ModusToolbox™ software	ModusToolbox™ software simplifies development for IoT designers. It delivers easy-to-use tools and a familiar microcontroller (MCU) integrated development environment (IDE) for Windows, macOS, and Linux
PSoC™ Creator	PSoC™ Creator enables concurrent hardware and firmware editing, compiling and debugging of PSoC™ devices. Applications are created using schematic capture and over 150 pre-verified, production-ready peripheral Components. Look in the downloads tab for Quick Start and user guides
Peripheral Driver Library (PDL)	Installed by PSoC™ Creator 4.3 and available through GitHub. Visit mtb-pdl-cat1 ; for PSoC™ Creator, look in <PDL install folder>/doc for the User Guide and API Reference
Hardware Abstraction Layer (HAL)	Available through GitHub only. Visit mtb-hal-cat1

~~5 PSoC™ 6 application notes~~

~~DRAFT~~ 5.10.6 Revision history

Document version	Date of release	Description of changes
**	2017-02-16	New Application Note
*A	2017-03-08	Updated Template
*B	2017-06-09	Updated text and screen shots for release versions of PSoC™ Creator 4.1 and PDL 3.0.0 Other miscellaneous edits
*C	2017-08-23	Minor edits Ported to new application note document template Confidential tag removed
*D	2018-03-07	Added a new Figure 2 Updated Figure 4 and associated kit device part number Updated Figures 6 and 9 for PSoC™ Creator 4.2 beta 2 Emphasized using CM0+ as a support CPU for tasks such as BLE and CAPSENSE™ Added references to AN221111, Creating a Secure System; AN217666, PSoC™ 6 Interrupts; AN219528, PSoC™ 6 Low Power Modes; and CE216795, PSoC™ 6 Dual-CPU Updated power modes description Miscellaneous minor edits Ported to new application note template Changed the document title to PSoC™ 6 MCU Dual-CPU System Design
*E	2018-06-11	Expanded section 4.3 to include dual-CPU debugging with µVision and IAR Embedded Workbench IDEs Updated power mode descriptions in section 3 Miscellaneous minor edits Ported to *Y application note template
*F	2018-06-12	Added support for ModusToolbox™
*G	2020-05-25	Updated instructions based on ModusToolbox™ 2.1 Added HAL support
*H	2020-08-26	Updated instructions based on ModusToolbox™ 2.2
*I	2022-09-19	Migrated to Infineon template Updated instruction based on ModusToolbox™ 3.0

5.11 AN215671 PSoC™ 6 MCU firmware design for BLE applications

About this document

•
1
1

Scope and purpose

AN215671 introduces the Bluetooth® low energy (BLE) stack used with PSoC™ 6 MCU with Bluetooth® low energy (BLE) connectivity, the BLE component architecture, and the use of a dual-CPU processor for BLE applications.

5 PSoC™ 6 application notes

This application note shows how to design firmware for BLE applications. This application note uses the associated code examples to demonstrate the multi-master multi-slave feature of PSoC™ 6 BLE and validate the application using the CySmart™ BLE host emulation tool and Cypress' iOS/Android CySmart™ application.

To access an ever-growing list of hundreds of PSoC™ code examples, please visit our [code examples web page](#). You can also explore the Cypress video training library [here](#).

5 PSoC™ 6 application notes~~DO NOT USE~~
5.11.1 Introduction

Bluetooth® low energy (BLE) is a low-power wireless standard introduced by the Bluetooth™ special interest group (SIG) for short-range communication. The BLE physical layer, protocol stack, and profile architecture are designed and optimized to minimize power consumption. Similar to classic Bluetooth™, BLE operates in the 2.4 GHz ISM band with a maximum bandwidth of 2 Mbps.

PSoC™ 6 MCU with BLE connectivity (PSoC™ 6 BLE) is Infineon ultra-low-power PSoC™ device with dual CPUs specifically designed for wearables and internet of things (IoT) products. PSoC™ 6 BLE integrates a BLE 4.2 radio and a royalty-free protocol stack with enhanced security and privacy; it also has throughput compliant with the BLE 5.0 specification.

This application note introduces you to the BLE protocol stack in PSoC™ 6 BLE and discusses how to develop BLE applications using the PSoC™ creator BLE component. The BLE component has built-in support for the standard profiles defined by Bluetooth® SIG, which simplifies application development. In addition, this application note uses the code examples [CE223508](#) and [CE224714](#) to demonstrate the multi-master multi-slave feature of PSoC™ 6 BLE and validate the application using the CySmart™ BLE host emulation tool and Cypress' iOS/Android CySmart™ application.

This application note assumes that you are familiar with the basics of BLE, CySmart 6 BLE, and the PSoC™ creator IDE. If you are new to PSoC™ 6 BLE, refer to [AN210781](#) – getting started with PSoC™ 6 MCU with Bluetooth® low energy (BLE) connectivity. To get acquainted with PSoC™ creator IDE, see the [PSoC™ Creator home page](#).

~~DRAFT~~

5 PSoC™ 6 application notes

5.11.2 PSoC™ resources

Infineon provides a wealth of data at www.infineon.com to help you to select the right PSoC™ device and quickly and effectively integrate it into your design. The following is an abbreviated list of resources for PSoC™ 6 MCU:

- Overview: [PSoC™ Portfolio](#), [PSoC™ Roadmap](#)
- Product selectors: [PSoC™ 6 MCU](#)
- [Datasheets](#) describe and provide electrical specifications for each device family
- [Application Notes](#) and [Code Examples](#) cover a broad range of topics, from basic to advanced level. Many of the application notes include code examples. You can also browse our collection of code examples from directly inside PSoC™ creator
- [Technical Reference Manuals \(TRMs\)](#) provide detailed descriptions of the architecture and registers in each device family
- [PSoC™ 6 MCU Programming Specifications](#) provide the information necessary to program the nonvolatile memory of PSoC™ 6 MCU devices
- [CAPSENSE™ Design Guides](#): Learn how to design capacitive touch-sensing applications with PSoC™ devices
- Development Tools

[CY8CKIT-062-BLE PSoC™ 6 BLE Pioneer Kit](#) is an easy-to-use and inexpensive development platform for PSoC™ 6 BLE

- Training Videos: Cypress provides [video training](#) on our products and tools, including a dedicated series on [PSoC™ 6 MCU](#)

See [KBA223067](#) for a comprehensive list of PSoC™ 6 MCU resources.

~~DRAFT~~ 5 PSoC™ 6 application notes

5.11.3 BLE protocol implementation in PSoC™ 6 BLE

The BLE protocol stack used with PSoC™ 6 BLE integrates a low-energy controller and host, and provides a full and flexible API for developing BLE applications. The BLE stack consists of two major blocks: BLE host and BLE controller. Partitioning the stack into host and controller provides flexibility of implementing the protocol stack using the dual-CPU architecture of PSoC™ 6 BLE. Figure 443 shows the dual-CPU and single-CPU BLE implementation in PSoC™ 6 BLE.

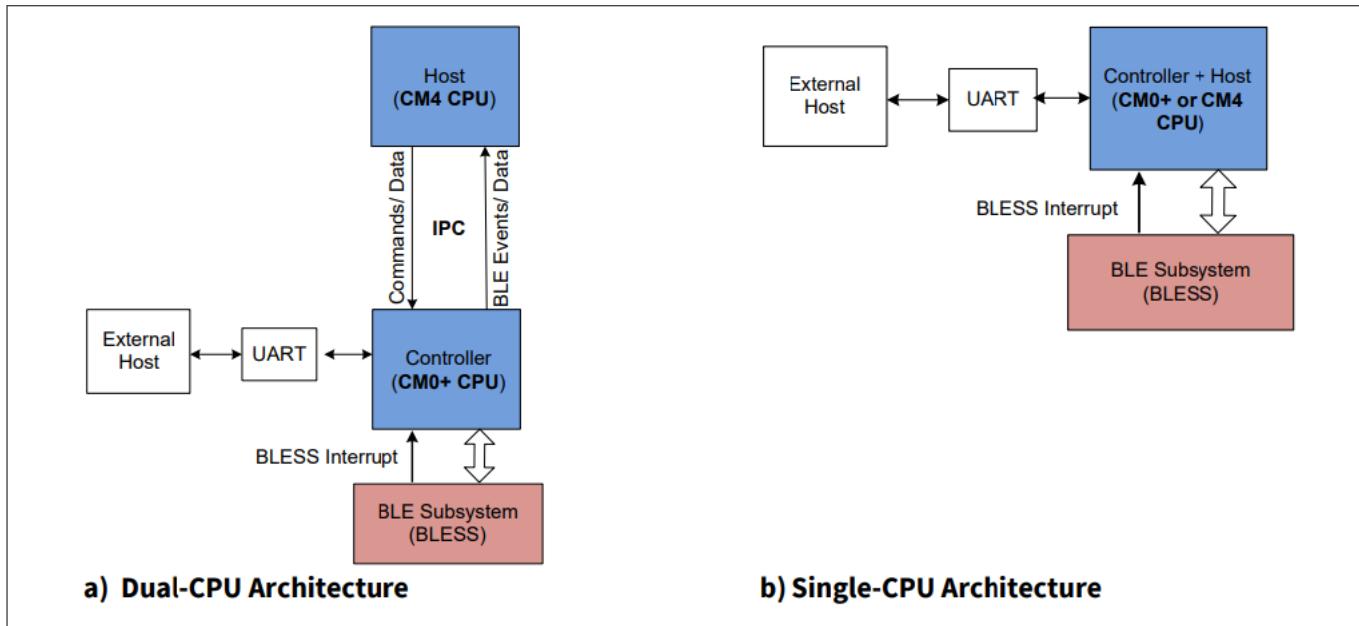


Figure 443 **BLE Architecture in PSoC™ 6**

The BLE stack is provided as a precompiled library along with the Cypress BLE middleware library, and packaged with Cypress peripheral driver library (PDL) 3.x. The BLE middleware library contains a comprehensive API that allows you to configure the BLE stack and the underlying hardware.

The BLE stack has the following features:

- Operates in 2.4 GHz ISM band with a data rate of 2 Mbps, compliant with the BLE 5.0 specification
- Multi-master multi-slave (MMMS) feature: supports up to four simultaneous connections in any combination of roles (central/peripheral)
- Simultaneous support for all generic access profile roles – peripheral, central, broadcaster, and observer
- Attribute protocol (ATT) that defines how the application data is organized and accessed
- Generic attribute profile (GATT) that defines methods to access data defined by the ATT layer (GATT server and GATT client)
- Support for BLE special interest group (SIG)-adopted GATT-based profiles and services
- Security manager protocol (SMP) that provides a toolbox for secure data exchange over the BLE link. This toolbox includes:
 - Pairing methods: just works, passkey entry, out of band, numeric comparison
 - Authenticated man-in-the-middle (MITM) protection and data signing
- Logical link control and adaptation protocol (L2CAP) connection-oriented channel
- Link layer (LL) features which include:
 - Master and slave role
 - 128-bit AES encryption
 - Low duty cycle advertising
 - LE (low energy) ping

5 PSoC™ 6 application notes

The BLE stack implements a layered architecture of the BLE protocol as shown in [Figure 444](#).

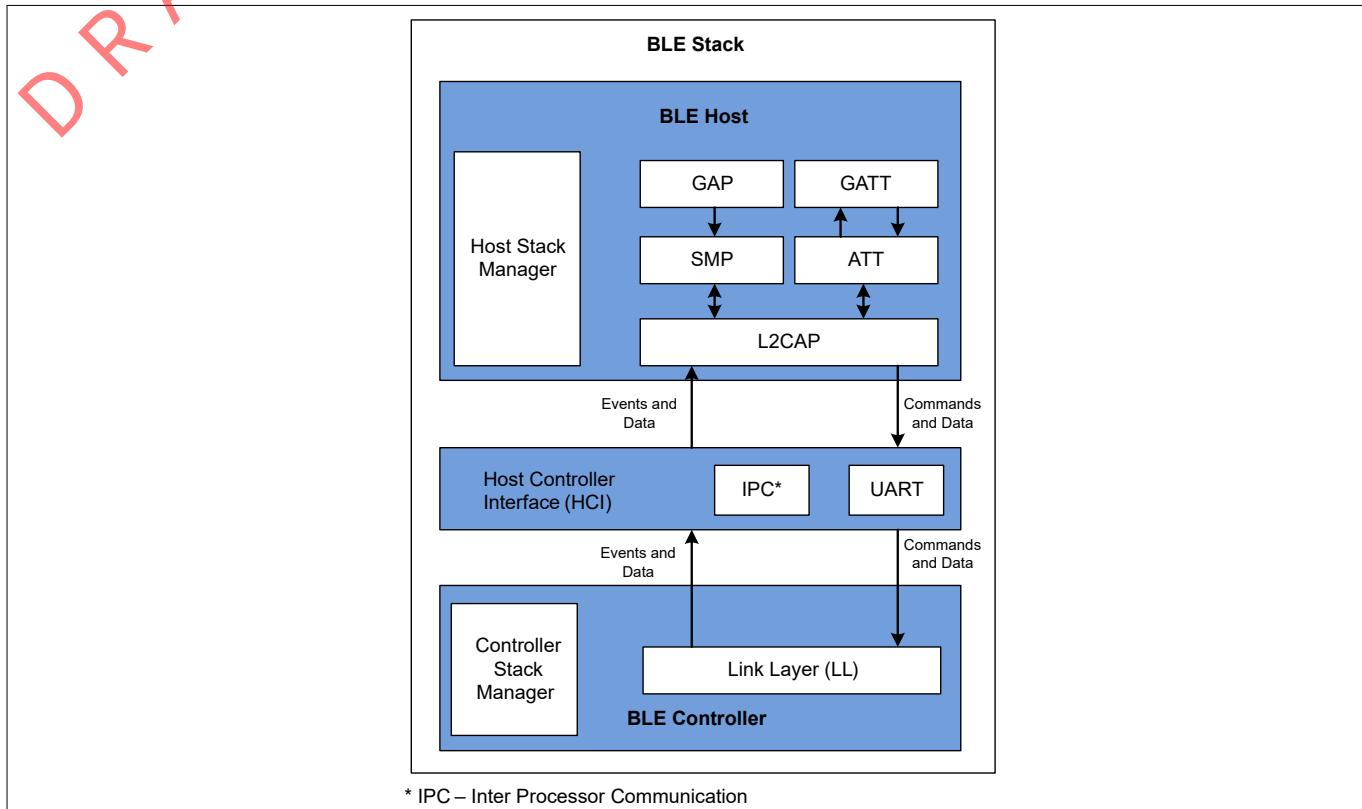


Figure 444 BLE protocol stack

5.11.3.1 BLE host

The BLE host implements the following protocol layers:

- Generic access profile (GAP)
- Generic attribute profile (GATT)
- Attribute protocol
- Security manager protocol (SMP)
- Logical link control and adaptation protocol (L2CAP)
- Host controller interface (HCI)

You do not need a full working knowledge of the complex BLE protocol to develop a BLE application. From an application point of view, it is sufficient to have a good understanding of the following:

- GAP layer: This layer deals with how the BLE link is established between two devices
- GATT layer: This layer deals with how the data is represented; GATT is the layer where the data values are transferred

In the remainder of the section, you will get an overview of GATT and GAP protocols and the related terms. You may refer to the [Bluetooth® Core Specification](#) for a detailed description of the protocol.

5.11.3.1.1 Generic access profile (GAP)

GAP is the lowest layer of the BLE stack that an application interface with. It includes parameters that govern advertising and connection, among other things. To establish a BLE link between two devices (Cypress BLE pioneer kit and a smartphone, for example), you need to understand the two GAP device roles:

~~5 PSoC™ 6 application notes~~

- **GAP central:** The GAP central is the device that initiates the connection with a peer device (GAP peripheral). A GAP central device scans for advertisements from GAP peripherals and establishes a connection with them. A smartphone that connects to a heart-rate measurement device is an example of a GAP central device
- **GAP peripheral:** The GAP peripheral is the device that is connected to a GAP central device. A GAP peripheral advertises its presence and accepts the connection from a GAP central device

The Bluetooth® core specification defines observer and broadcaster roles in addition to the central and peripheral roles. Observers listen to what's happening on the air; broadcasters send but do not receive information.

Note: Irrespective of the GAP role, the device at either end of a BLE link is referred to as a peer device.

Depending upon the GAP role of the device, the device can be in any of the following states:

- **Advertising:** To establish a BLE link between two peer devices, the GAP peripheral must advertise to let central devices know of its presence. It sends out advertising packets at timed intervals, known as the advertising interval, that ranges from 20 ms to 10.24 s. The advertisement interval determines how fast a link is established.
An advertisement packet can contain up to 31 bytes of data, which consist of the local device name, information pertaining to the services the device contains, etc. When a central device receives an advertisement packet, if the central device is configured as an active scanner, it may optionally send a request for more device-related data, called a scan request. The peripheral responds to the request by sending a scan response that can contain an additional 31 bytes
- **Scanning:** Scanning is used by the GAP central to listen to advertisement packets from peer devices. In this context, you need to consider two parameters: Scan window and scan interval. The central device starts a scan once per scan interval. Within the interval, it will scan for the duration of the scan window. If the scan window is equal to the scan interval, the central device does a continuous scan
- **Initiating:** When the central receives an advertisement packet, if it wants to establish a connection, it sends a connection request. This procedure is called “initiating”
- **Connected:** The central and peripheral are in a connected state from the first data exchange. When connected, the central requests data from the peripheral at a specifically defined interval called the “connection interval”. The connection interval is dictated by the central when the link is established. A peripheral can send connection parameter update requests to the central. The connection interval must be between 7.5 ms and 4 ms per the Bluetooth® core specification.

The link layer (LL) maintains the BLE link by sending at least one BLE packet in every connection interval. If the peripheral does not respond to packets from the central within the time frame, called “connection supervision timeout”, the link is considered lost. If the peripheral has no data to send and power consumption is of concern in your design, then it can choose to ignore a certain number of intervals. The number of ignored intervals is called the “slave latency”.

For a proper operation, it is recommended to have connection supervision timeout as follows:

$$\text{Connection supervision timeout} \geq (1 + \text{slave latency}) \times \text{connection interval}$$

5.11.3.1.2 Generic attribute profile (GATT)

After the central device establishes a connection with the peripheral, both devices are said to be connected over a BLE link. On a connected BLE link, independent of the GAP role, the generic attribute profile (GATT) defines two profile roles based on the source and destination of the data:

- **GATT server:** A GATT server is a device that contains the data or state. When configured by a GATT client, it sends data to the GATT client or modifies its local state. For example, a heart-rate measurement device is a GATT server that sends heart-rate data to a smartphone, which acts as the GATT client. Similarly, a smart

~~5 PSoC™ 6 application notes~~

bulb is a GATT server that contains the state of the bulb (ON/OFF) that can be configured by a smart switch, which acts as the GATT client

- **GATT client:** A GATT client is a device that configures the state of a GATT server or receives data from a GATT Server. For example, a smartphone that receives heart-rate information from the heart-rate measurement device is a GATT client

5.11.3.1.3 ATT protocol – organizing the data

The GATT server uses the ATT protocol, which organizes the data systematically in the form of an attribute table called GATT database. A GATT server uses attributes, characteristics, and services to represent and abstract data in a BLE device. A service contains one or more characteristics; each characteristic is composed of multiple attributes that contain the actual data. [Figure 445](#) shows the GATT database hierarchy.

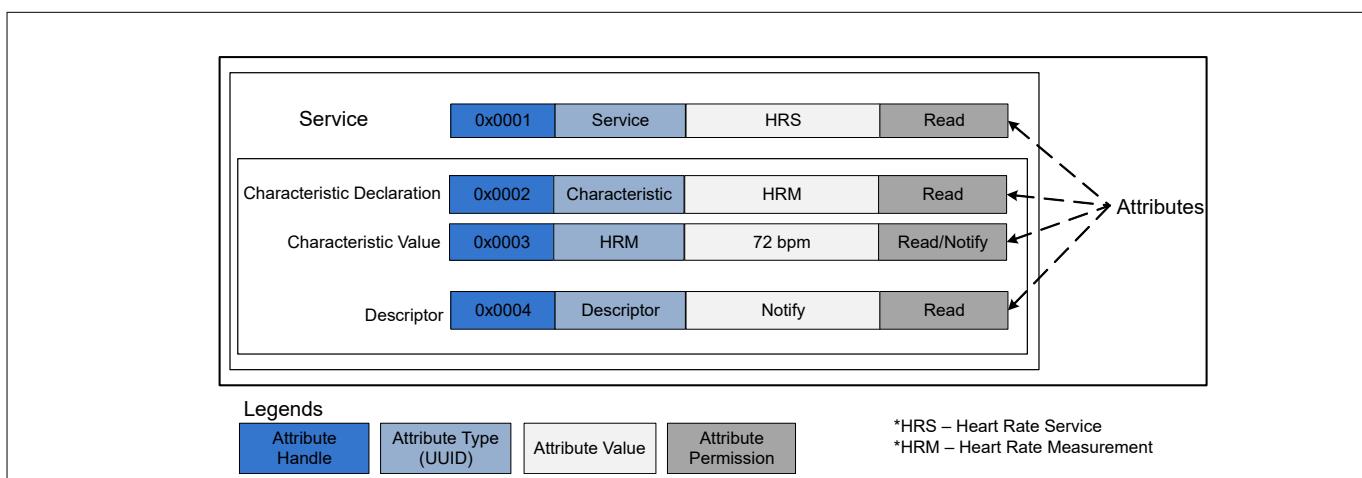


Figure 445 GATT database overview

Attribute: An attribute is the fundamental data container of the GATT layer that represents a discrete piece of information. The structure of an attribute consists of the following:

- Attribute handle: Used to address the attribute. A handle is a unique index of the attribute in the GATT database
- Attribute type: A 16-bit universally unique identifier (UUID) assigned by the Bluetooth® SIG that specifies the data contained in the attribute
- Attribute value: Contains the actual data
- Attribute permission: Specifies read/write permissions and security requirements for the attribute

Characteristic: A characteristic consists of at least two attributes: A characteristic declaration and an attribute that holds the value for the characteristic. All data that will be transferred through a GATT service must be mapped to a set of characteristics. It is a good idea to bundle the data so that each characteristic is a self-contained, single-instance data point. For example, if some pieces of data always change together, it will often make sense to collect them in one characteristic. The characteristic illustrated in [Figure 445](#) has three attributes: Characteristic declaration, characteristic value, and the descriptor attribute.

Descriptors: Any attribute within a characteristic definition other than characteristic declaration attribute and characteristic value attribute is a descriptor. A descriptor is an additional attribute that provides more information about a characteristic, for instance, a human-readable description of the characteristic.

However, there is one special descriptor that is worth mentioning in particular: The client characteristic configuration descriptor (CCCD). This descriptor is added for any characteristic that supports the notify or indicate properties. Writing a '1' to the CCCD enables notifications, while writing a '2' enables indications. Writing a '0' disables both notifications and indications. Notify and Indicate are the attribute properties that allow a GATT server to make the GATT client aware of the changes to a characteristic. The indicate property has application-level acknowledgment while notify does not have application-level acknowledgment.

5 PSoC™ 6 application notes

~~DRAFT~~

Service: A service is composed of one or more related characteristics that define a function or feature of a device. GATT services typically include pieces of related functionality such as a sensor's readings and settings, or the inputs and outputs of a human interface device.

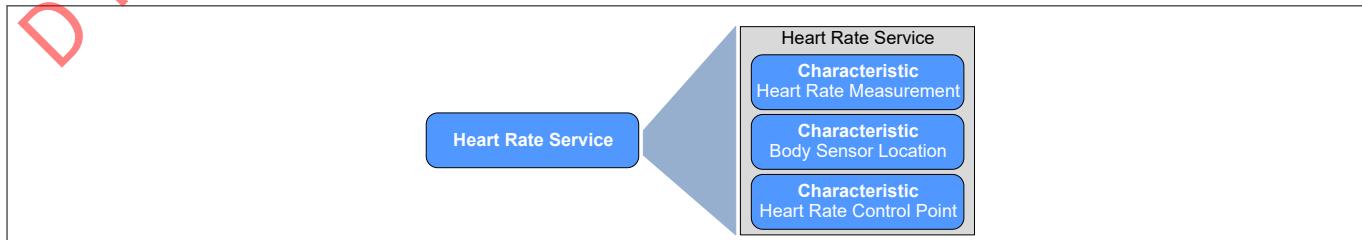


Figure 446 GATT service example

Profile: A BLE profile is a specification that guarantees application-level interoperability between profile-compliant devices. It defines the role and configuration of different BLE layers and GATT service(s) to be supported to create a specific end application or use case. For example, in the case of a heart-rate monitoring device, the BLE heart rate profile defines the required GAP and GATT roles, and the GATT services to be supported by the heart-rate monitoring device to create an interoperable heart-rate monitoring device. The Bluetooth® SIG offers a set of predefined standard profiles for commonly used BLE end applications. In addition, you can create your own custom profiles that consist of standard or custom services.

The profile defines two application roles:

- **Sensor or server:** The sensor profile role is supported by the application that has data. The sensor profile specification defines the required roles (for example, GATT/GAP roles) and behavior (for example, advertisement interval, GATT services to be supported) of the BLE device to support a sensor application use case. Following the sensor profile specification guarantees interoperability of the sensor application with any other device that implements the corresponding collector profile. For example, a heart-rate monitoring device that implements the heart rate sensor profile will be interoperable with all smartphones that implement the heart rate collector profile
- **Collector or client:** The collector profile role is supported by the application that wants data. The collector profile specification defines the required BLE device roles and behavior to interoperate with and collect information from any device that implements the corresponding sensor profile specification

Figure 447 illustrates the data abstraction and hierarchy in a BLE device.

5 PSoC™ 6 application notes

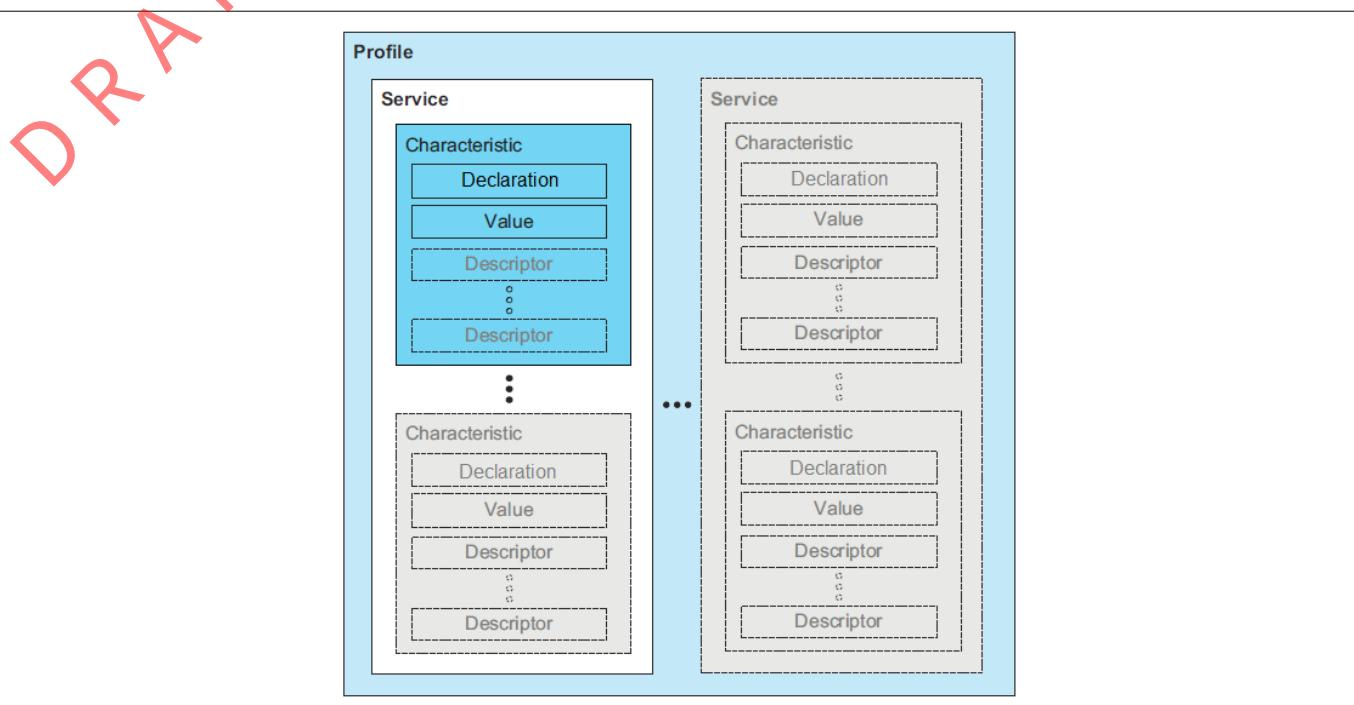


Figure 447 **BLE data hierarchy³⁴⁾**

Figure 448 illustrates the application-level interaction of two connected BLE devices using the GAP and GATT protocol layers.

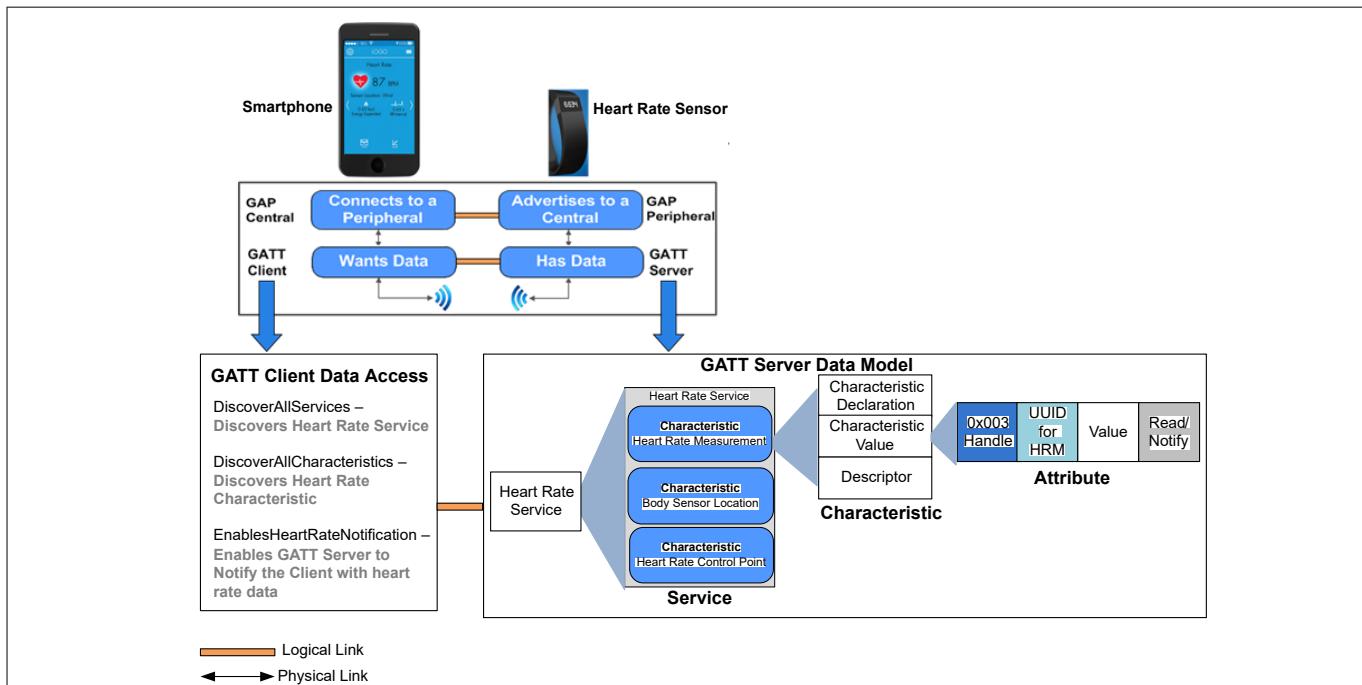


Figure 448 **BLE application level overview**

5.11.3.2 BLE controller

The BLE controller implements the link layer (LL) of the BLE protocol. In PSoC™ 6 BLE, LL is a hardware-firmware co-implementation. The LL firmware implements a state machine that manages and controls the physical BLE

³⁴ Image courtesy of Bluetooth® SIG

~~5 PSoC™ 6 application notes~~

connections between devices. It supports all LL states such as advertising, scanning, initiating, and connecting. It implements all the key link control procedures such as LE encryption, LE connection update, LE channel update, and LE ping.

~~5.11.3.2.1 Managing multiple connections~~

The PSoC™ BLE stack may have up to four instances of the LL to support the multiple connection feature. Each LL instance caters to a particular BLE link and manages the LL functionality such as advertising, scanning, initiating, and connecting, independent of other BLE connection links.

During a connection event, the BLE stack assigns an identification number called “bdHandle” to the connecting peer device. A BLE event called CY_BLE_EVT_GAP_DEVICE_CONNECTED is triggered with an event parameter containing the bdHandle. During an active BLE connection, the application identifies the connected peer device using the bdHandle. [Figure 449](#) illustrates a simplified block diagram on managing multiple BLE connections using PSoC™ 6 BLE.

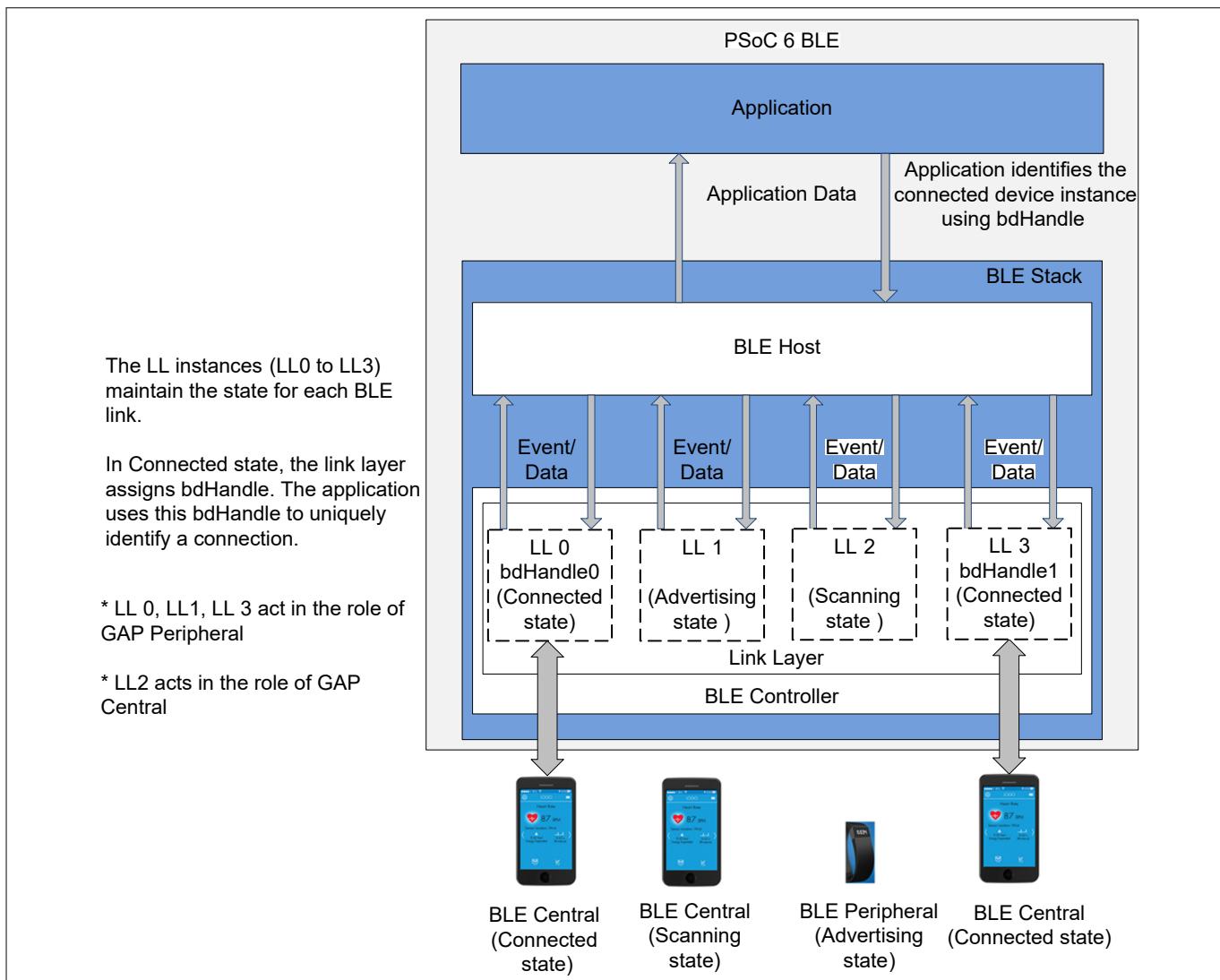


Figure 449 Multiple BLE connections using PSoC™ 6 BLE

This section described how a BLE link is established using the GAP layer and how the data is represented using the GATT layer. With this background, let us examine the typical firmware flow for developing a BLE application.

5 PSoC™ 6 application notes

~~DRAFT~~ 5.11.4 Developing a BLE application: Firmware low

This section provides an overview of the minimal steps to be followed while developing BLE applications using PSoC™ 6 BLE. The BLE component in PSoC™ creator abstracts the BLE protocol into a simple and easy-to-use GUI. [Figure 450](#) shows a high-level architecture of the BLE component, which illustrates the relationship between the BLE stack layer and the route in which the application interacts with the component.

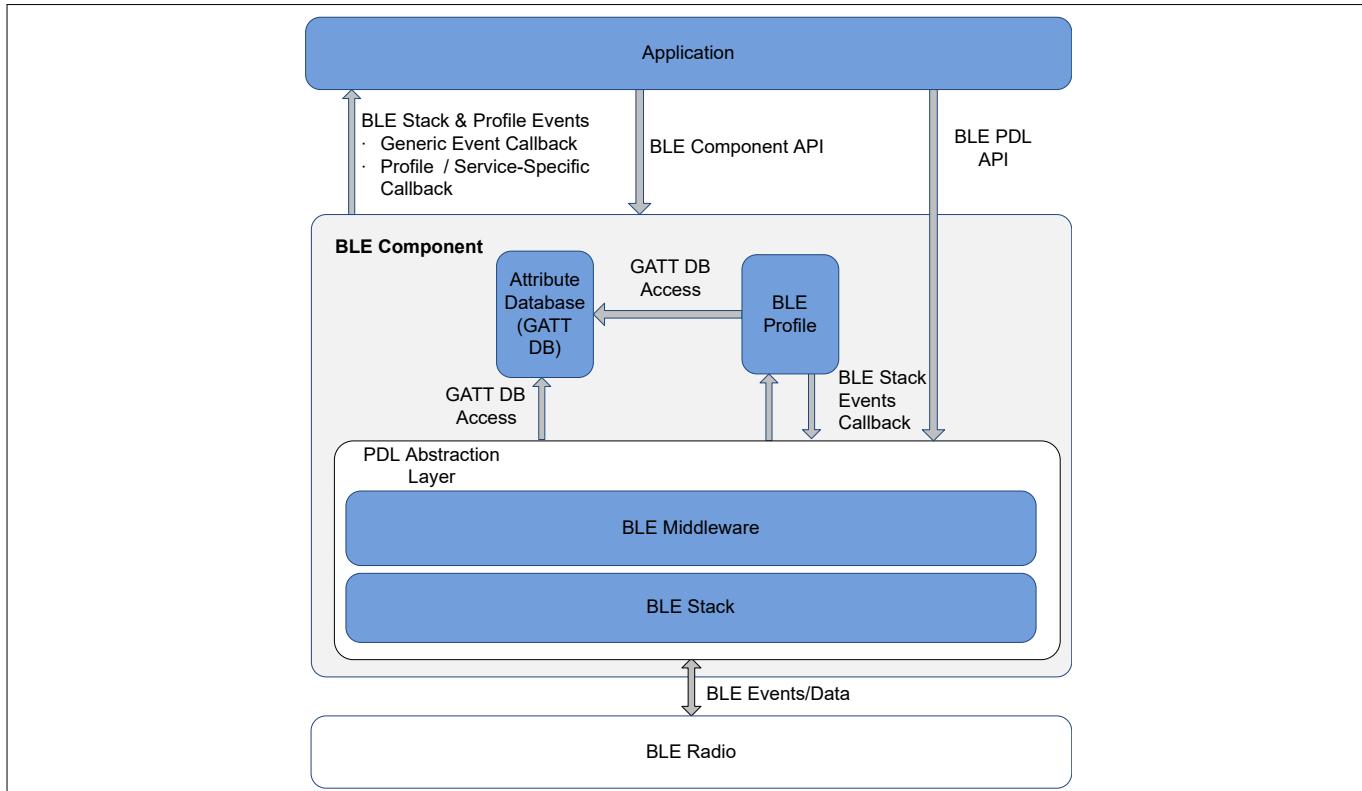


Figure 450 **BLE component architecture**

There are two major steps for developing a BLE application with PSoC™ 6 BLE:

- Configure the BLE component
- Write the firmware

The following sections provide the details on how to accomplish each task.

• **Configure the BLE component**

- **Select the GAP role and number of simultaneous connections**

PSoC™ 6 BLE supports up to four simultaneous connections in any combinations of GAP roles (central, peripheral, observer, broadcaster). In this step, you configure the desired GAP roles and the number of simultaneous BLE connections

- **Select the BLE architecture**

PSoC™ 6 BLE offers dual CPU architecture, where the controller runs on the CM0+ CPU and host runs on CM4 CPU, and single CPU architecture, where both controller and host are run only on one CPU (either CM0+ or CM4). [Figure 451](#) illustrates this sub steps A and B

5 PSoC™ 6 application notes

DRAFT

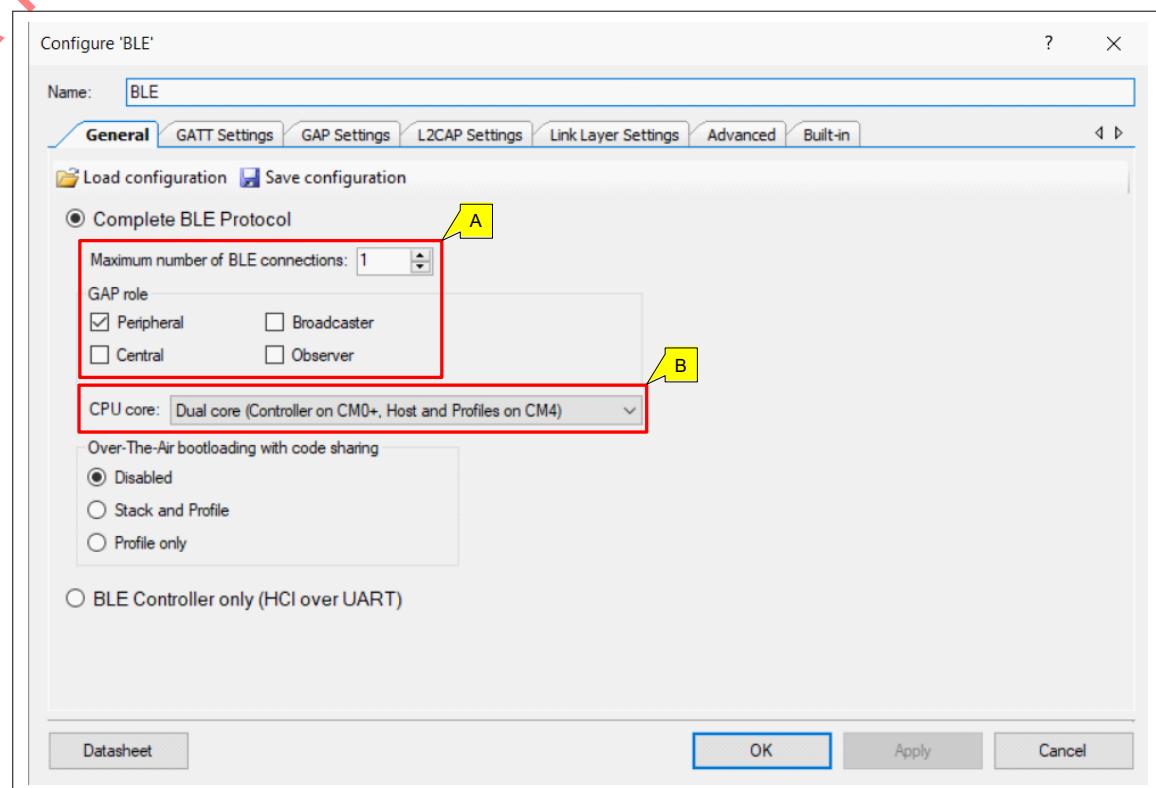


Figure 451 General BLE settings

Assign the BLE subsystem (BLESS) interrupt to the CPU on which the BLE controller runs. In PSoC™ creator, assign the BLESS interrupt to the controller CPU in the Interrupts tab of the design wide resources window as shown in Figure 452. By default, the BLESS interrupt is assigned to the CM0+ CPU.

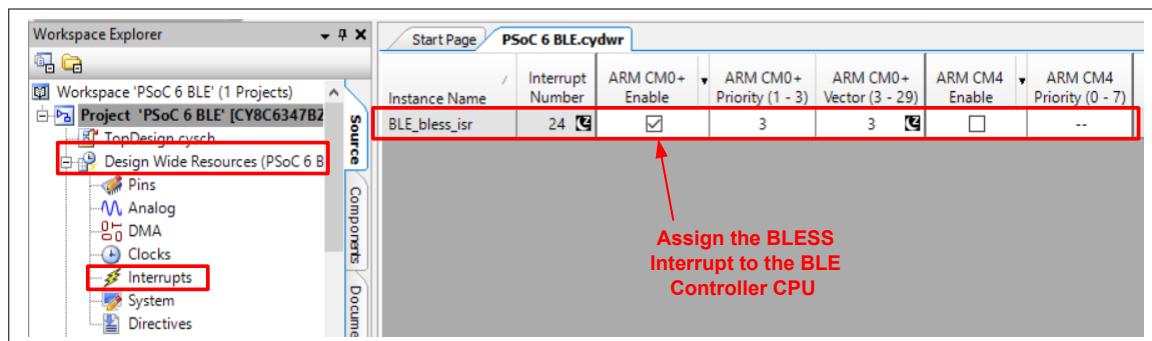


Figure 452 Assigning the BLESS interrupt to the BLE controller CPU

Note: The rest of the document assumes dual-CPU architecture for the BLE application.

- Configure GAP settings

Depending upon the GAP role of the device, you must configure a few settings specific to the role. For example, if the device is a GAP peripheral, then you must provide the advertisement settings, advertisement packet structure, etc. Similarly, if it is GAP central, you need to provide the scan setting and connection parameters. If the device has multiple roles, you need to provide the configuration settings for each GAP role. Figure 453 to Figure 457 show GAP configuration.

5 PSoC™ 6 application notes

DRAFT

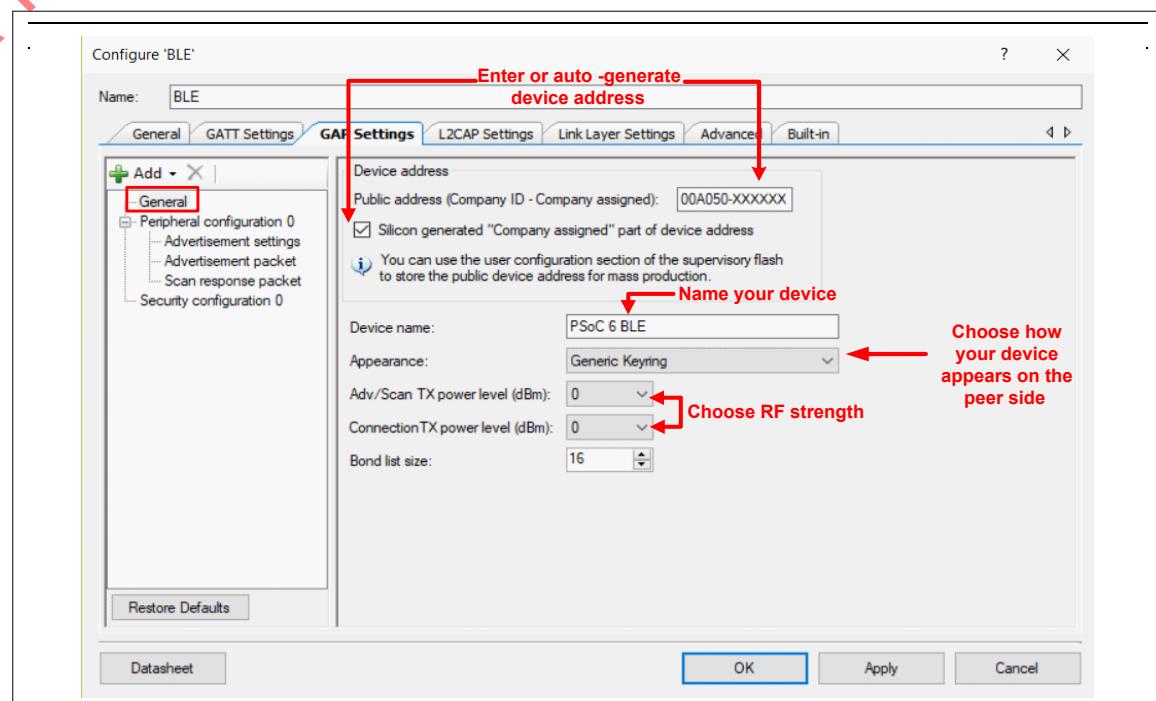


Figure 453 GAP general settings

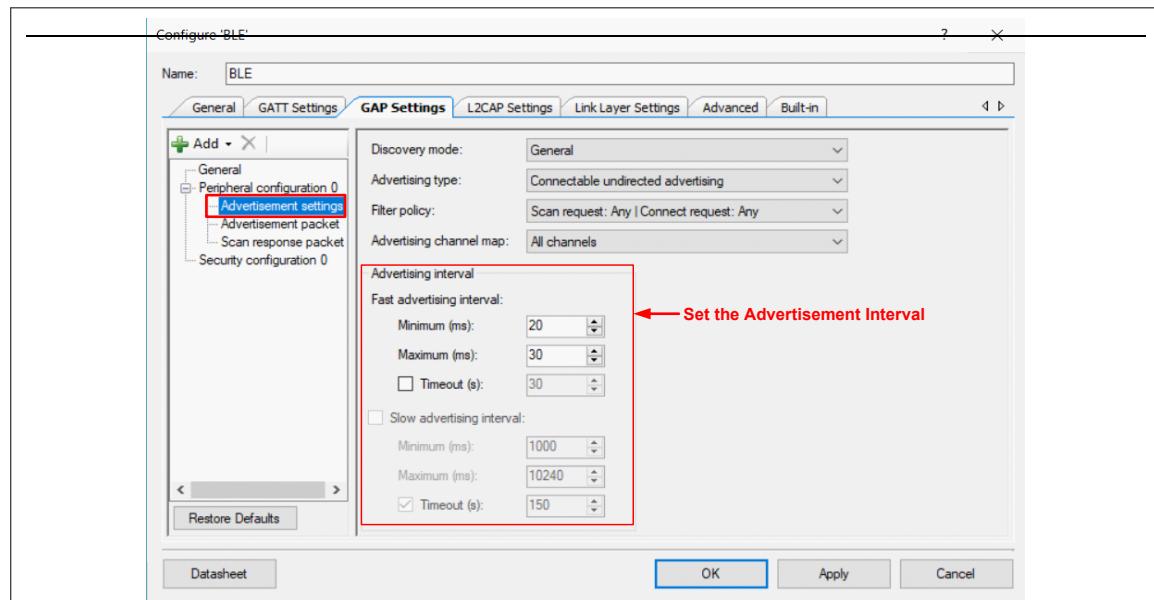


Figure 454 GAP peripheral configuration: Advertisement settings

5 PSoC™ 6 application notes

DRAFT

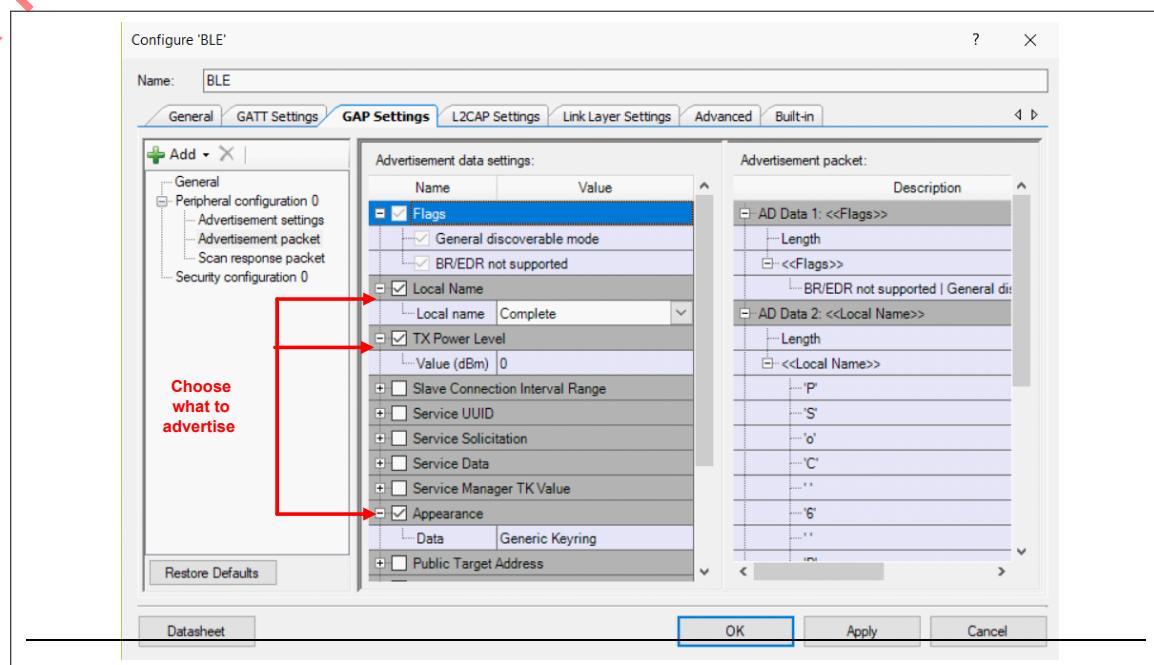


Figure 455 GAP peripheral configuration: Advertisement packet

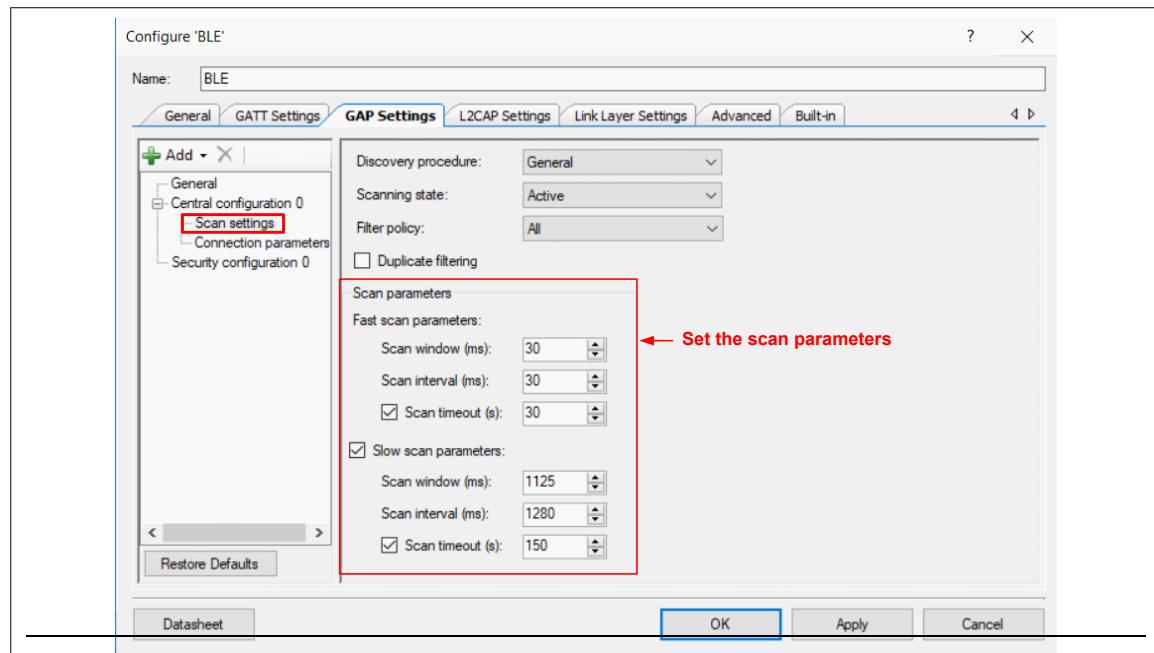


Figure 456 GAP central configuration: Scan settings

5 PSoC™ 6 application notes

DRAFT

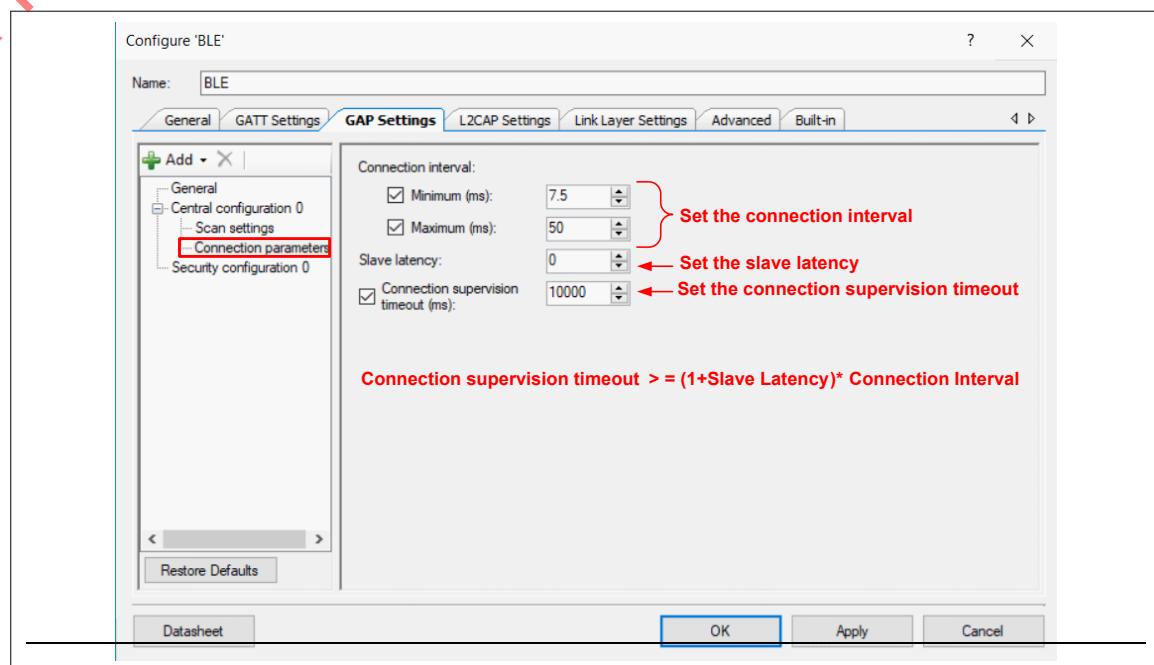


Figure 457 GAP central configuration: Connection parameters

The parameters in GAP configuration settings are explained in chapter [Chapter 5.11.3.1.1](#). For a detailed description of each parameter, see the [BLE Component datasheet](#). Further, examples provided in chapter [Section 5: BLE Design Examples](#) illustrate how these parameters are set.

- Configure GATT settings

In this step, you include the services that you need in your application. The BLE component supports numerous Bluetooth® SIG-adopted GATT-based profiles and services. The component generates all the necessary code for a profile/service operation as configured in the component customizer. [Figure 458](#) shows an example for configuring a standard heart rate service (HRS).

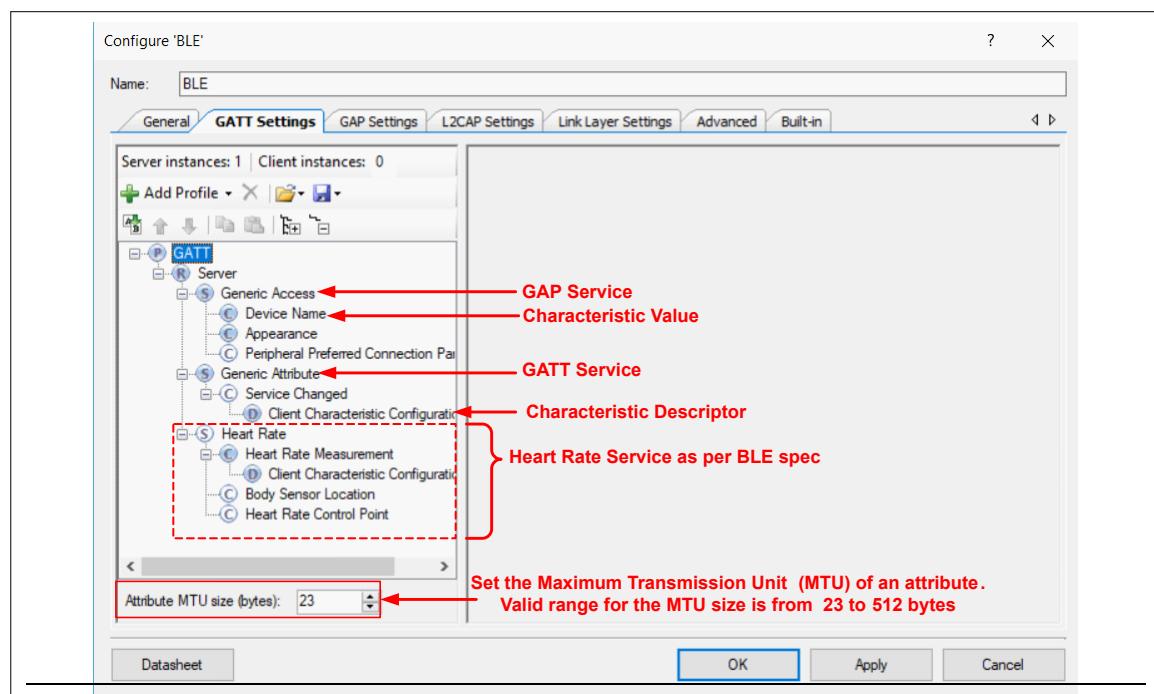


Figure 458 BLE component GATT configuration

5 PSoC™ 6 application notes

This completes all the necessary steps for GAP and GATT profile-based settings. After you configure the BLE component, the next step is to write the firmware to initialize the design and register event handler functions to process the BLE events and data as required by your application

• Write the firmware

This section discusses the typical steps required by a BLE application firmware. For each major task, specific steps are explained along with sample firmware code. For detailed firmware steps, refer to the code examples referenced in [Section 5: BLE Design Examples](#).

- Firmware flow for the controller (CM0+ CPU for dual-CPU architecture)

1. Start the BLE controller
2. Process BLE events using the `Cy_BLE_ProcessEvents()` function

The `Cy_BLE_ProcessEvents()` function checks the internal task queue in the BLE stack and processes any pending BLE operations. As shown in [Figure 459](#), this function must be called at least once every ‘T’ intervals, where ‘T’ is equal to the connection interval or scan interval (if the device is a GAP central), or the connection interval or advertisement interval (if the device is a GAP peripheral) whichever is smaller.

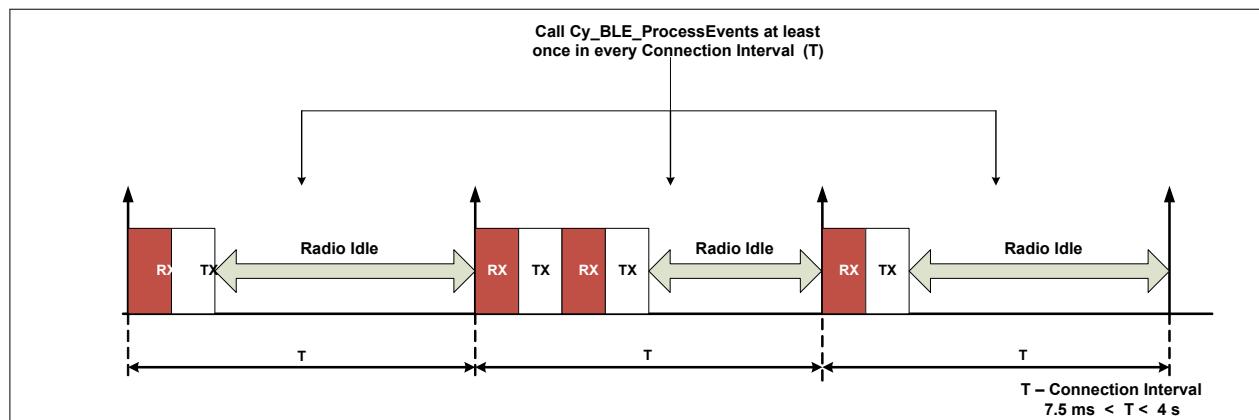


Figure 459 **BLE connection interval**

5 PSoC™ 6 application notes

The following code snippet shows the basic controller firmware flow:

```
#include "project.h"

int main(void)
{
    /* Enable global interrupts */
    __enable_irq();

    /* Start the controller portion of BLE. Host runs on the CM4 */
    if(Cy_BLE_Start(NULL) == CY_BLE_SUCCESS)
    {
        /* Enable CM4 only if BLE Controller started successfully.
           CY_CORTEX_M4_APPL_ADDR must be updated if CM4 memory layout
           is changed. */

        Cy_SysEnableCM4(CY_CORTEX_M4_APPL_ADDR);

    }
    else
    {
        /* Halt the CPU */
        CY_ASSERT(0u);

    }

    for(;;)
    {
        /* Process the controller portion of the BLE events and wake up the host (CM4) and send
        data to the host via IPC if necessary */

        Cy_BLE_ProcessEvents();

        /* Put CM0+ to Deep Sleep mode. The BLE hardware automatically wakes
        up the CPU if processing is required */

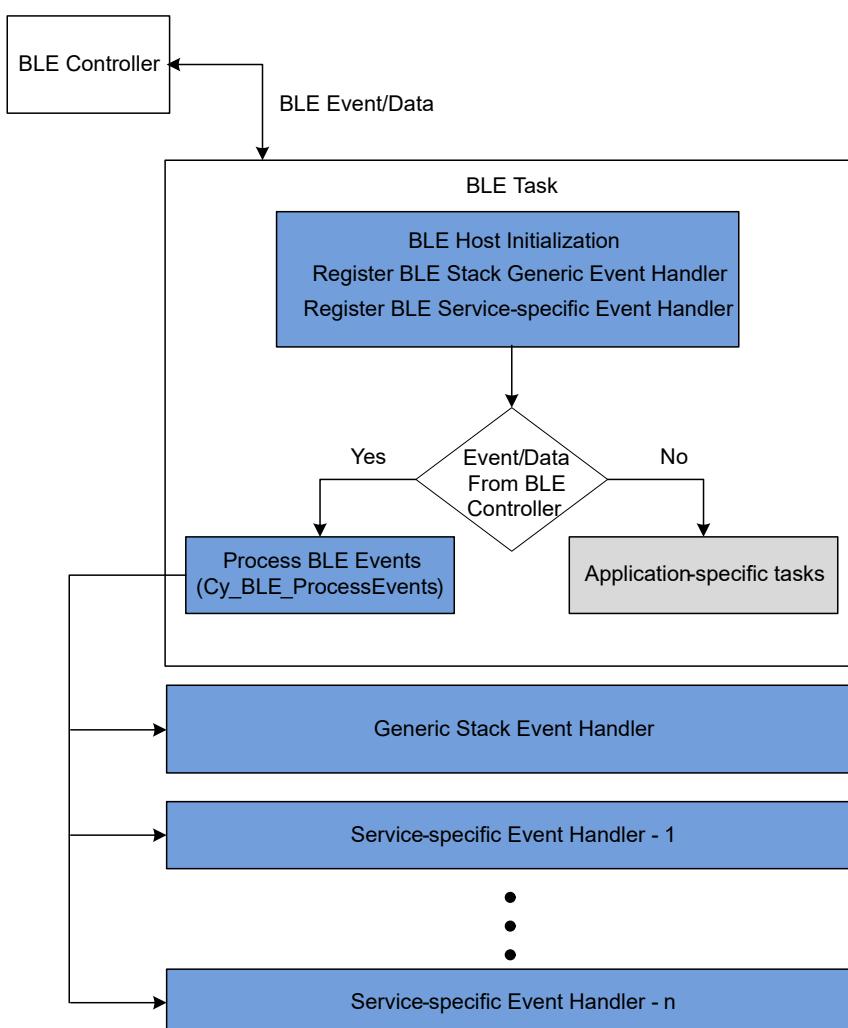
        Cy_SysPm_DeepSleep(CY_SYSPM_WAIT_FOR_INTERRUPT);
    }
}/*End of main function */
```

- Firmware flow for the host (CM4 CPU for dual-CPU architecture)

The interaction between BLE controller and the BLE host are event-driven. The BLE stack generates events that provide status and data to the application firmware running on the BLE Host

5 PSoC™ 6 application notes

DRAFT



The typical firmware flow for the host is as follows:

- Register the application host callback function
- The application host callback function is called when the BLE host needs to process pending stack events. The application host callback function processes the pending events by calling `Cy_BLE_ProcessEvents()`. Note that the application host callback function executes from within an ISR and must be very short.

5 PSoC™ 6 application notes

- Start the BLE host by providing the generic BLE generic event handler function
- Register BLE service-specific event handlers

```

void BleAppHost_Callback(void)
{
    /* On every interrupt from the BLE Controller, process the pending
       Host events propagated from the BLE Controller */
    Cy_BLE_ProcessEvents();
}

int main(void)
{
    __enable_irq(); /* Enable global interrupts. */

    /* Start BLE component and register generic event handler */
    if(Cy_BLE_Start(GenericStackEventHandler) == CY_BLE_SUCCESS)
    {

        /* Register the Host application callback function */
        Cy_BLE_RegisterAppHostCallback(BleAppHost_Callback);

        /* Register Service Specific event handler*/
        Cy_BLE_HRS_RegisterAttrCallback(HRSEventHandler);
    }
    else
    {
        /* Halt the CPU */
        CY_ASSERT(0u);
    }

    for(;;)
    {
        /* Code specific to your application */
    }
}

```

BLE events are handled with the user-defined generic BLE stack event handler. In the above code snippet, `GenericStackEventHandler()` is a user-defined function to handle BLE events. The generic stack event handler must handle a few basic events from the stack. [Table 85](#) lists these events.

Table 85 Basic BLE stack events

BLE stack event name	Event description	Event handler action
CY_BLE_EVT_STACK_ON	BLE stack initialization is completed successfully	GAP central: Discover GAP peripherals using <code>Cy_BLE_GAPC_StartScan()</code> . GAP peripheral: Start the advertisement using <code>Cy_BLE_GAPP_StartAdvertisement()</code> .

(table continues...)

5 PSoC™ 6 application notes

~~DRAFT~~ **Table 85** (continued) Basic BLE stack events

BLE stack event name	Event description	Event handler action
CY_BLE_EVT_GAP_DEVICE_CONNECTED	BLE link with the peer device is established	Application-specific action
CY_BLE_EVT_GAPP_ADVERTISEMENT_START_STOP	BLE stack advertisement start/stop event	Application-specific action
CY_BLE_EVT_GAP_DEVICE_DISCONNECTED	BLE link with the peer device is disconnected	GAP central: Discover GAP Peripherals using Cy_BLE_GAPC_StartScan(). GAP peripheral: Start the advertisement using Cy_BLE_GAPP_StartAdvertisement().
CY_BLE_EVT_HARDWARE_ERROR	BLE hardware error	Application-specific action
CY_BLE_EVT_STACK_SHUTDOWN_COMPLETE	BLE stack has been shut down	Application-specific action

The BLE Middleware Library provides functions for registering event handlers specific to standard BLE services. For example, `Cy_BLE_HRS_RegisterAttrCallback(HRSEventHandler)` registers the `HRSEventHandler` function to handle events specific to the heart rate service.

Code examples described in the [Section 5: BLE Design Examples](#) explain how to define and implement the generic event handler function and BLE service-specific event handlers. [Table 86](#) lists commonly used functions in a typical BLE design.

For a comprehensive list of BLE stack events and functions, see the Cypress BLE middleware library. To open the documentation, from PSoC™ creator, navigate to **Help > Documentation > Peripheral Driver Library**. Locate the **Cypress BLE middleware library** under the **Middleware** menu option.

Table 86 Basic BLE API functions

BLE API Function	Description
<code>Cy_BLE_Start()</code>	Initializes the BLE stack and initializes the profile layer, schedulers, timers, and other platform-related resources required by the BLE component
<code>Cy_BLE_GAPP_StartAdvertisement()</code>	GAP peripherals use this function to start the advertisement using the advertisement data set in the BLE component
<code>Cy_BLE_GAPP_StopAdvertisement()</code>	GAP peripherals use this function to stop advertisement
<code>Cy_BLE_GAPC_StartScan()</code>	GAP central uses this function to discover GAP peripherals that are available for connection
<code>Cy_BLE_GAPC_StopScan()</code>	GAP central uses this function to stop the discovery of GAP peripherals
<code>Cy_BLE_GAPC_ConnectDevice()</code>	GAP central uses this function to send connection requests to the GAP peripheral with the connection parameters set in the BLE component
<code>Cy_BLE_ProcessEvents()</code>	Checks the internal task queue in the BLE stack and processes pending BLE events

5 PSoC™ 6 application notes

5.11.4.1 ~~DRAFT~~ Implementing low-power BLE design

Low-power operation is one of the major considerations when it comes to connected devices. PSoC™ 6 BLE provides low-power modes without sacrificing the performance. PSoC™ 6 BLE has several power modes that can affect either the whole system or just a single CPU. CPU power modes are active, sleep, and deep sleep as defined by Arm®. Device system power modes are low power (LP), ultra low power (ULP), deep sleep, and hibernate.

[AN219528 – PSoC™ 6 MCU Low-Power/modes and Power Reduction Techniques](#) describes the low-power design considerations and the low-power modes in PSoC™ 6 MCU. In this section, we shall see how to implement a low-power design involving BLE connectivity. For low-power support, configure the BLE Component as shown in [Figure 460](#).

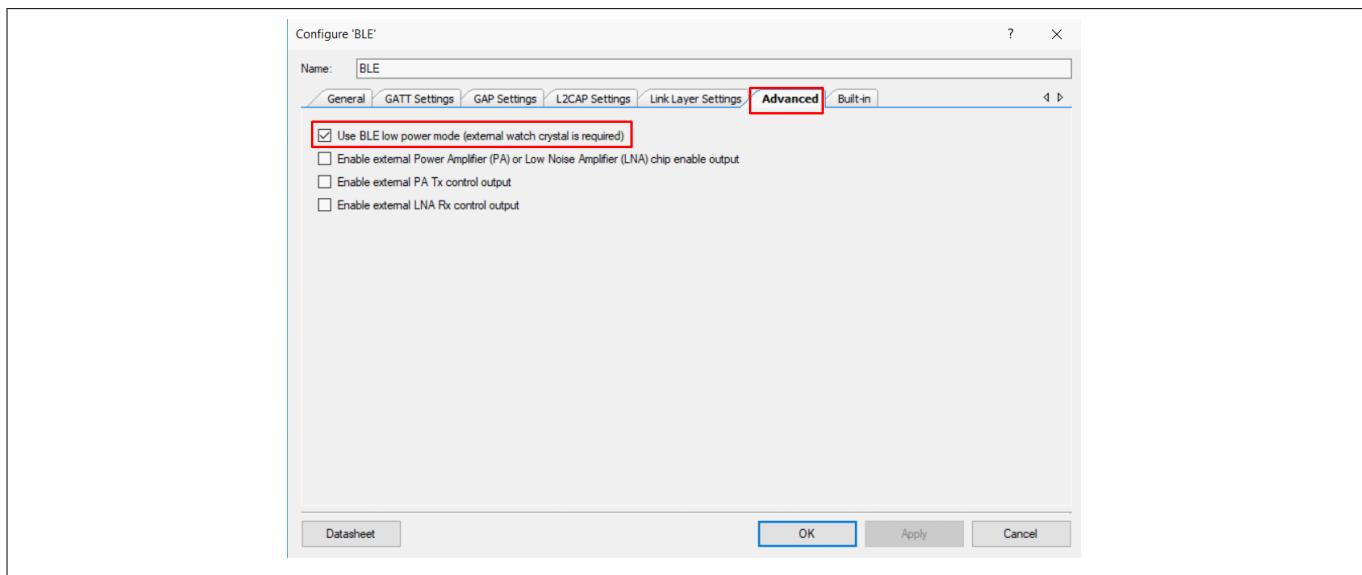


Figure 460 Configuring BLE low-power support

To operate the BLE component in low-power mode, it is mandatory to use an external watch crystal oscillator (WCO) as the source to the low-frequency clock (LFCLK) of PSoC™ 6 BLE. [Figure 461](#) shows configuring the WCO as the LFCLK source in the design wide resources.

5 PSoC™ 6 application notes

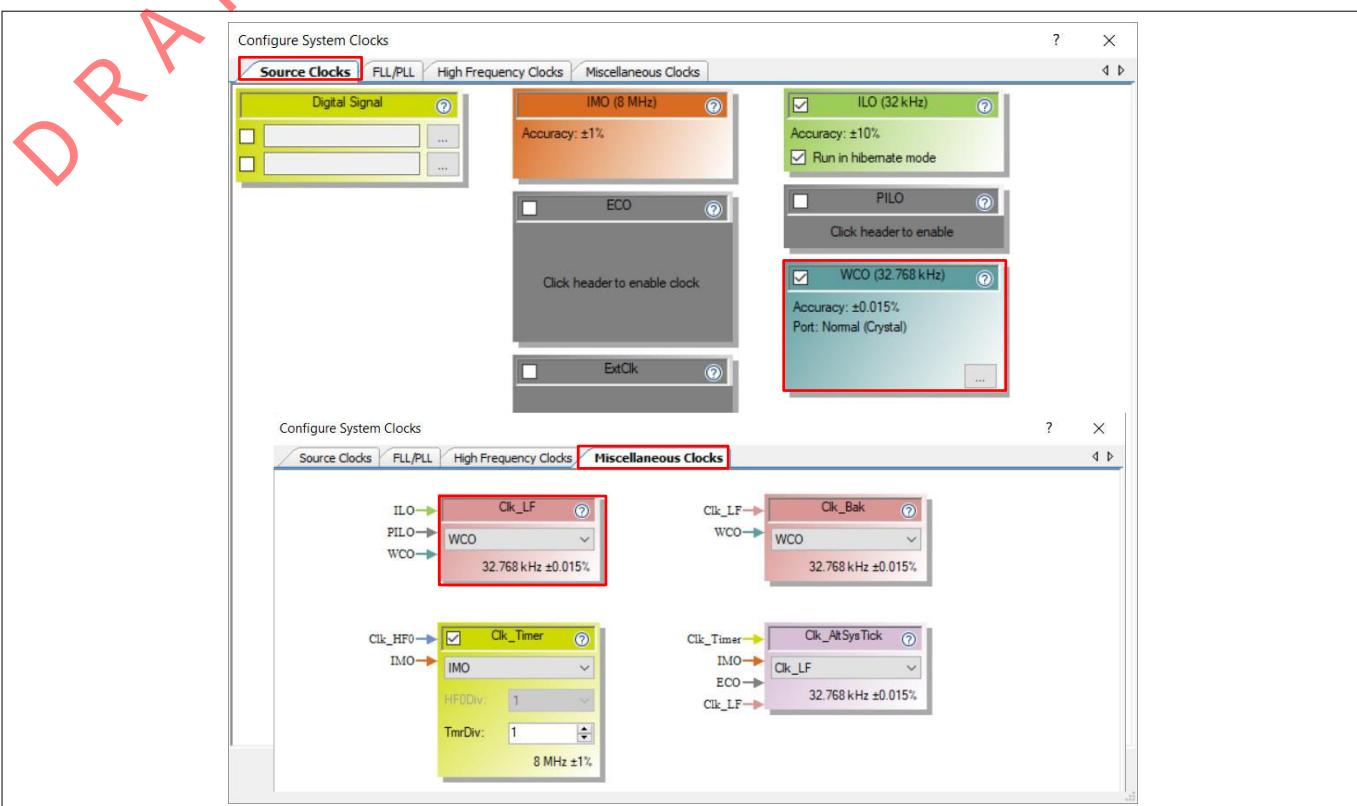


Figure 461 Enabling WCO for low-power operation

In PSoC™ 6 BLE, wakeup from the low-power mode operation is possible with a set of interrupts handled by the wakeup interrupt controller (WIC) block of PSoC™ 6 BLE. The WIC block supports up to 41 interrupts that can wake up a CPU from the CPU deep sleep power mode. Refer to [AN217666 – PSoC™ 6 MCU Interrupts](#) for more details on PSoC™ 6 MCU interrupts.

The BLESS interrupt is a CPU deep sleep-capable interrupt source. As described previously, BLE is implemented based on a host-controller architecture. The discussion that follows is split into two parts based on the BLE architecture: The dual-CPU architecture and the single-CPU architecture.

Dual-CPU architecture

The BLESS interrupt is mapped to the BLE controller CPU. The host CPU and the controller CPU can independently go into low-power mode if there are no BLE events to be processed by them. The following sequence of events is involved in a typical low-power BLE design for dual-CPU architecture:

- The BLESS interrupt wakes up the controller during every connection interval
- The controller CPU then services the BLE event with `Cy_BLE_ProcessEvents()` in firmware
- If the host must process a BLE event or data, the controller wakes up the host CPU using the inter-processor communication (IPC) interrupt
- Upon receiving the BLE event or data through the IPC interrupt, the host CPU processes it using `Cy_BLE_ProcessEvents()` and registered event handler functions. If the host CPU requires the service of the controller CPU, the host sends the BLE data/event using IPC

Note: *The BLE Component uses system IPC pipes for communication between the Host and the Controller. System IPC pipes and associated interrupts are automatically configured by PSoC™ Creator.*

5 PSoC™ 6 application notes

- ~~1.~~ The host CPU goes back to the low-power mode and waits for the IPC interrupt from the controller with the Cy_SysPm_DeepSleep(CY_SYSPM_WAIT_FOR_INTERRUPT) function
- ~~2.~~ The controller processes the BLE events/data from the Host (if any). The controller CPU goes back to the low power mode and waits for the BLESS interrupt using the Cy_SysPm_DeepSleep(CY_SYSPM_WAIT_FOR_INTERRUPT) function

Single-CPU Architecture

In single-CPU architecture, both the Controller and the Host runs on the same CPU (CM0+ or CM4). The CPU used by the BLE Component is hereafter referred to as the BLE CPU for the sake of explanation. The following sequence of events is involved in a typical low-power BLE design for single-CPU architecture:

- The BLESS interrupt wakes up the BLE CPU
- The Cy_BLE_ProcessEvents() function processes both the controller and the Host BLE events/data
- The BLE CPU goes back to the low power mode and wait for the BLESS interrupt using the Cy_SysPm_DeepSleep(CY_SYSPM_WAIT_FOR_INTERRUPT) function

Note: When the design enables BLE low-power mode as shown in [Figure 460](#), the BLE component registers the low-power callback function. When the low-power mode transition is about to occur using the cy_SysPm_DeepSleep() function, the registered callback function is executed. The CPU enters low-power mode only if the BLE stack is ready for low-power operation. If the design involves other blocks capable of low-power operation, your application must define how power mode transitions occur. See 'System Power Management' in the PDL API Reference Manual.

5 PSoC™ 6 application notes

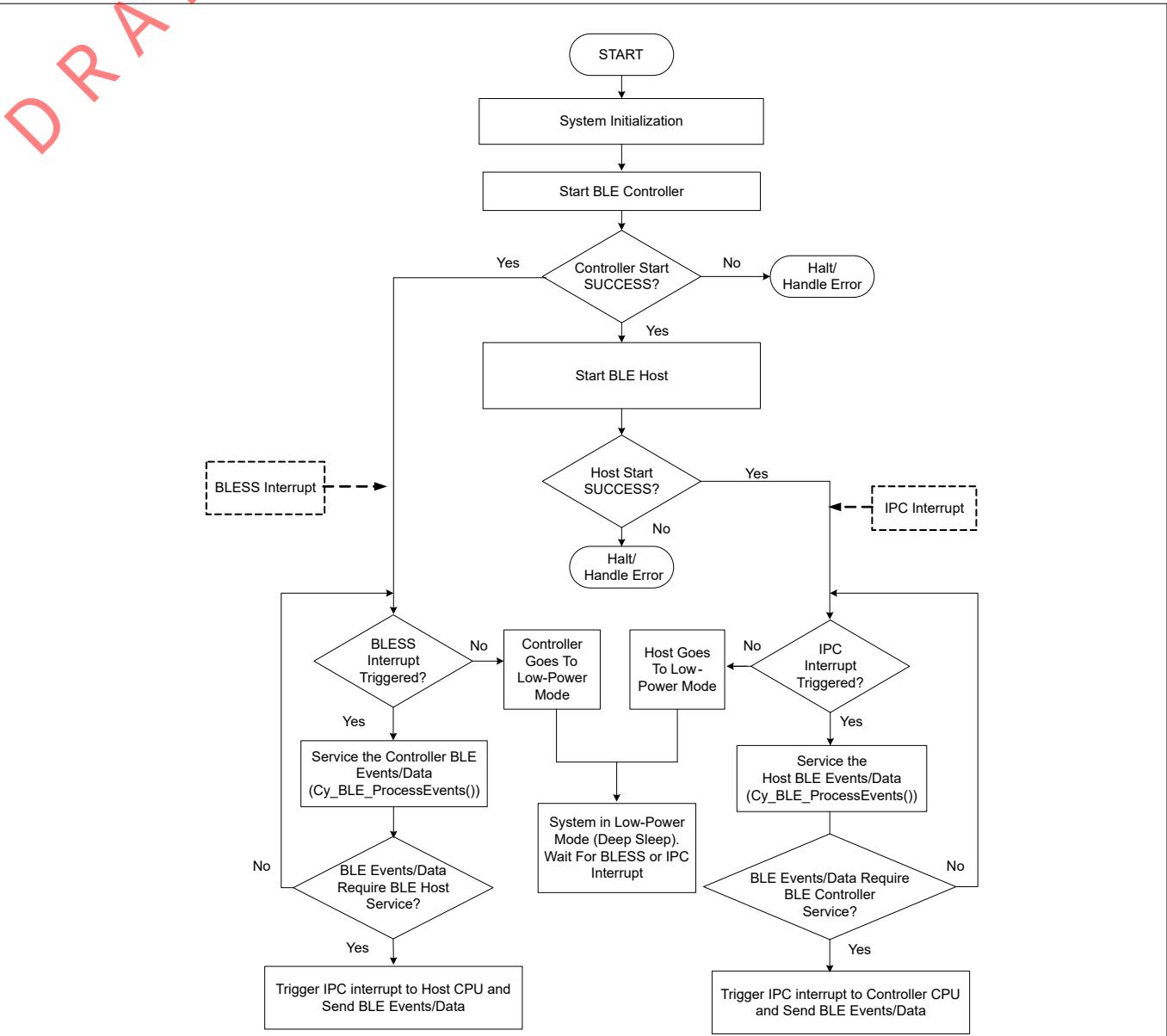


Figure 462 Typical firmware flow for low-power BLE design

5.11.4.2 Implementing a secure BLE design

For every wireless device, protecting a user's private data is of paramount significance. PSoC™ 6 BLE is compliant with the Bluetooth® low energy (BLE) 4.2 security features. The BLE stack in PSoC™ 6 BLE supports the security manager protocol (SMP) with the following features:

- Encryption and authentication of user data
- Authenticated man-in-the-middle (MITM) protection and data signing
- Support for device bonding and different pairing methods such as just works, passkey entry, out of band, and numeric comparison
- Pairing method selection based on the I/O capability of the GAP central and the GAP peripheral devices

This section discusses how to configure the BLE component to incorporate the security features into your design and handle them in firmware. For technical details on the BLE 4.2 security features, see the [Bluetooth® Core Specification version 4.2 Volume 3 Part H](#). For details on developing secure embedded system with PSoC™ 6 MCU, see [AN221111 – PSoC™ 6 MCU: Creating a Secure System](#).

~~5 PSoC™ 6 application notes~~

Before proceeding further on the BLE security features, two new terms need to be introduced. The security manager protocol (SMP) layer of the BLE stack defines two roles for a BLE device in establishing a secure connection. They are:

Initiator: The link layer master always initiates a secure connection. Therefore, a GAP central device is the initiator.

Responder: The link layer slave responds to the secure connection request from the link layer master; therefore, a GAP peripheral device is the responder.

5.11.4.2.1 Configuring security features using the BLE component security mode and security level

PSoC™ 6 BLE supports two modes of security with multiple security levels within each mode as [Table 87](#) shows.

Table 87 BLE security modes and levels

Security mode	Security level	Remark
Mode 1	No security (no authentication, no encryption)	This mode is used in designs where data encryption is required
	Unauthenticated pairing with encryption	
	Authenticated pairing with encryption	
	Authenticated LE secure connections pairing with encryption	
Mode 2	Unauthenticated pairing with data signing	This mode is used in designs where data signing is required
	Authenticated pairing with data signing	

I/O Capabilities

Security levels described in [Table 87](#) take effect based on the device's input and output capability. Every BLE device determines its peer device input and output capabilities during the pairing request phase explained in chapter [Establishing secure BLE link: Firmware flow](#). [Table 88](#) summarizes the device capabilities configurable from the BLE component:

Table 88 Device I/O Capability

I/O capabilities	Description	GAP authentication required (Yes/No)
Display	Used in devices with display capability and may display authentication data	Yes
Display Yes/No	Used in devices with display and at least two input keys for the Yes/No action	Yes
Keyboard	Used in devices with numeric keypad	Yes
No Input No Output	Used in devices that do not have any capability to enter or display the authentication key data to the user	No
Keyboard and Display	Used in devices like PCs and tablets	Yes

5 PSoC™ 6 application notes

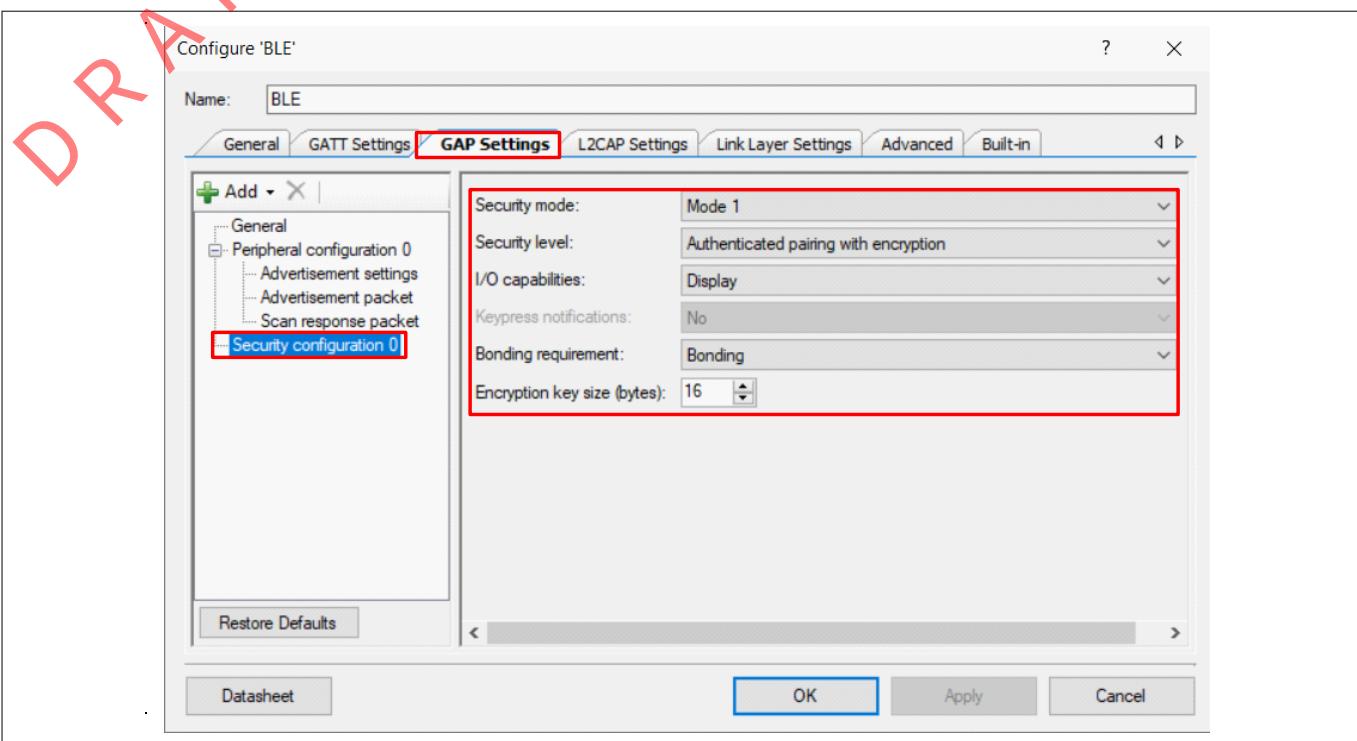


Figure 463 Configuring BLE security settings

When two devices want to communicate securely with BLE, they need to follow authentication procedure called “pairing”. PSoC™ 6 BLE supports four pairing methods compliant with BLE 4.2 feature as follows:

- **Just works:** When the I/O capability of the device is set to No Input No Output and no MITM protection is required, the BLE device uses this pairing method
- **Passkey entry:** The user either inputs an identical passkey into both devices, or one device displays the passkey and the user enters that passkey into the other device
- **Numeric comparison:** Both devices display a six-digit number and the user authenticates by selecting ‘Yes’ if both devices display the same number
- **Out of band (OOB):** This pairing method is used when both devices have access to an out-of-band mechanism to discover the devices as well as exchange the information used in the pairing procedure

See [AN99209 – PSoC™ 4 BLE and PRoC BLE: Bluetooth® LE 4.2 Features](#) for an elaborate description on BLE pairing process and establishing a secure connection.

The pairing method is determined based on the device security mode and the I/O capability of the interacting BLE devices as summarized in [Table 89](#).

Table 89 Pairing method based on the I/O capabilities of interacting BLE devices

GAP role	Initiator					
Respon der	I/O capabilities	Display only	Display (Yes/No)	Keyboard only	No Input No Output	Keyboard and display
	Display only	Just works	Just works	Passkey entry	Just works	Passkey entry
	Display (Yes/No)	Just works	Numeric comparison	Passkey entry	Just works	Numeric comparison
	Keyboard only	Passkey entry	Passkey entry	Passkey entry	Just works	Passkey entry

(table continues...)

Reference manual

5 PSoC™ 6 application notes

DRAFT Table 89 (continued) Pairing method based on the I/O capabilities of interacting BLE devices

GAP role	Initiator	No Input No Output	Just works	Just works	Just works	Just works
	No Input No Output	Just works	Just works	Just works	Just works	Just works
	Keyboard and display	Passkey entry	Numeric comparison	Passkey entry	Just works	Numeric comparison

5.11.4.2.2 Establishing secure BLE link: Firmware flow

The following steps illustrate the firmware flow for establishing a secure BLE link:

- The pairing procedure requires generating a set of security keys. These keys need to be generated irrespective of the GAP role. The BLE device exchanges the generated key with the peer device during the key exchange stage of the authentication procedure. On the CY_BLE_EVT_STACK_ON BLE event, generate the keys using the Cy_BLE_GAP_GenerateKeys() function
 - Upon successful generation of security keys, the CY_BLE_EVT_GAP_KEYS_GEN_COMPLETE event is triggered. Set the device identity address using the Cy_BLE_GAP_SetIdAddress() function
 - On CY_BLE_EVT_GAP_DEVICE_CONNECTED or CY_BLE_EVT_GAP_ENHANCE_CONN_COMPLETE event, the BLE device sets the security keys using the Cy_BLE_GAP_SetSecurityKeys() function. The Link Layer of the BLE stack uses these keys for encrypting the data
 - The initiator sends a pairing request using Cy_BLE_GAP_AuthReq(). The parameter passed through this function contains the security mode/level, I/O capability, encryption key size, etc. This triggers the CY_BLE_EVT_GAP_AUTH_REQ BLE event at the GAP peripheral
 - The GAP peripheral (responder) sends a request to the GAP central device (Initiator) to initiate the pairing procedure using the cy_BLE_GAP_AuthReq() function. This triggers the CY_BLE_EVT_GAP_AUTH_REQ BLE event at the GAP central device
 - On receiving the CY_BLE_EVT_GAP_AUTH_REQ event, the GAP peripheral responds with Cy_BLE_GAPP_AuthReqReply(). The pairing response contains much of the same information as the Initiator pairing request parameter
 - On successful pairing information exchange, the BLE stack generates the CY_BLE_EVT_GAP_SMP_NEGOTIATED_AUTH_INFO event at both the GAP central and GAP peripheral devices
 - During the authentication procedure, the BLE stack generates one of the following events based on the BLE device's I/O capabilities and security modes:
 - CY_BLE_EVT_GAP_PASSKEY_DISPLAY_REQUEST
 - CY_BLE_EVT_GAP_KEYPRESS_NOTIFICATION
 - CY_BLE_EVT_GAP_NUMERIC_COMPARISON_REQUEST
 - On successful completion of the pairing procedure, the BLE stack generates the CY_BLE_EVT_GAP_AUTH_COMPLETE event
- If authentication fails, CY_BLE_EVT_GAP_AUTH_FAILED is generated. Typically, this event is handled by disconnecting the BLE link using the CY_BLE_GAP_Disconnect() function.

Table 90 lists the sequence of BLE stack events and the action to be taken when using the LE secure connection feature.

~~DO NOT PUBLISH~~

5 PSoC™ 6 application notes

Table 90 BLE events and corresponding actions for LE secure connections

BLE stack event(s)	Event handler action
CY_BLE_EVT_STACK_ON	Generate the security keys using Cy_BLE_GAP_GenerateKeys()
CY_BLE_EVT_GAP_KEYS_GEN_COMPLETE	Set the device identity address using Cy_BLE_GAP_SetIdAddress()
CY_BLE_EVT_GAP_DEVICE_CONNECTED or CY_BLE_EVT_GAP_ENHANCE_CONN_COMPLETE	Set the security keys using Cy_BLE_GAP_SetSecurityKeys() Additionally, if it is a GAP peripheral, send a request to the GAP central to initiate pairing using Cy_BLE_GAP_AuthReq()
CY_BLE_EVT_GAP_AUTH_REQ	GAP central: Initiate pairing using Cy_BLE_GAP_AuthReq() GAP peripheral: Respond to the pairing request using Cy_BLE_GAPP_AuthReqReply()
CY_BLE_EVT_GAP_SMP_NEGOTIATED_AUTH_INFO	Initiate application-specific action
CY_BLE_EVT_GAP_PASSKEY_DISPLAY_REQUEST	Display the 6-decimal-digit value extracted from the event parameter
CY_BLE_EVT_GAP_KEYPRESS_NOTIFICATION	The BLE stack generates this event when a keypress (secure connections) is received from the peer device
CY_BLE_EVT_GAP_NUMERIC_COMPARISON_REQUEST	This event indicates that the device must display a passkey during the secure connection pairing procedure Call Cy_BLE_GAP_AuthPassKeyReply() with valid parameters on receiving this event
CY_BLE_EVT_GAP_AUTH_COMPLETE	This event indicates that the authentication procedure is completed Initiate application-specific action
CY_BLE_EVT_GAP_AUTH_FAILED	This event indicates that the authentication process between two devices has failed Typically, upon receiving this event, the BLE device disconnects the BLE link using CY_BLE_GAP_Disconnect()

Table 91 list the API functions commonly used for establishing a secure BLE connection. For a comprehensive list of BLE events and API functions specific to LE secure connection, see Cypress BLE Middleware Library.

Table 91 API functions for LE secure connection

API	Description
Cy_BLE_GAP_SetSecurityKeys()	This function sets the security keys that are to be exchanged with a peer device during the key exchange stage of the authentication procedure and sets it in the BLE stack
Cy_BLE_GAP_GenerateKeys()	This function generates the security keys as per application requirement

(table continues...)

5 PSoC™ 6 application notes

Table 91 (continued) API functions for LE secure connection

API	Description
Cy_BLE_GAP_AuthReq()	This function starts the authentication/pairing procedure with the peer device
Cy_BLE_GAPP_AuthReqReply()	The GAP peripheral device uses this function to send the pairing response in the authentication/pairing procedure
Cy_BLE_GAP_FixAuthPassKey()	This function sets or clears a fixed passkey to be used during the authenticated pairing procedure

Device bonding

The pairing procedure explained above is required to authenticate a peer device. If bonding is enabled, the device has the provision to store the link key of a previously authenticated link. If the bonding data is stored, and then if a connection is lost and re-established, bonded devices can resume the communication without the need of authentication again.

[Table 92](#) lists the API functions to handle device bonding in the firmware. [Figure 464](#) shows the flowchart for handling the pairing and bonding procedure in a secure BLE design.

5 PSoC™ 6 application notes

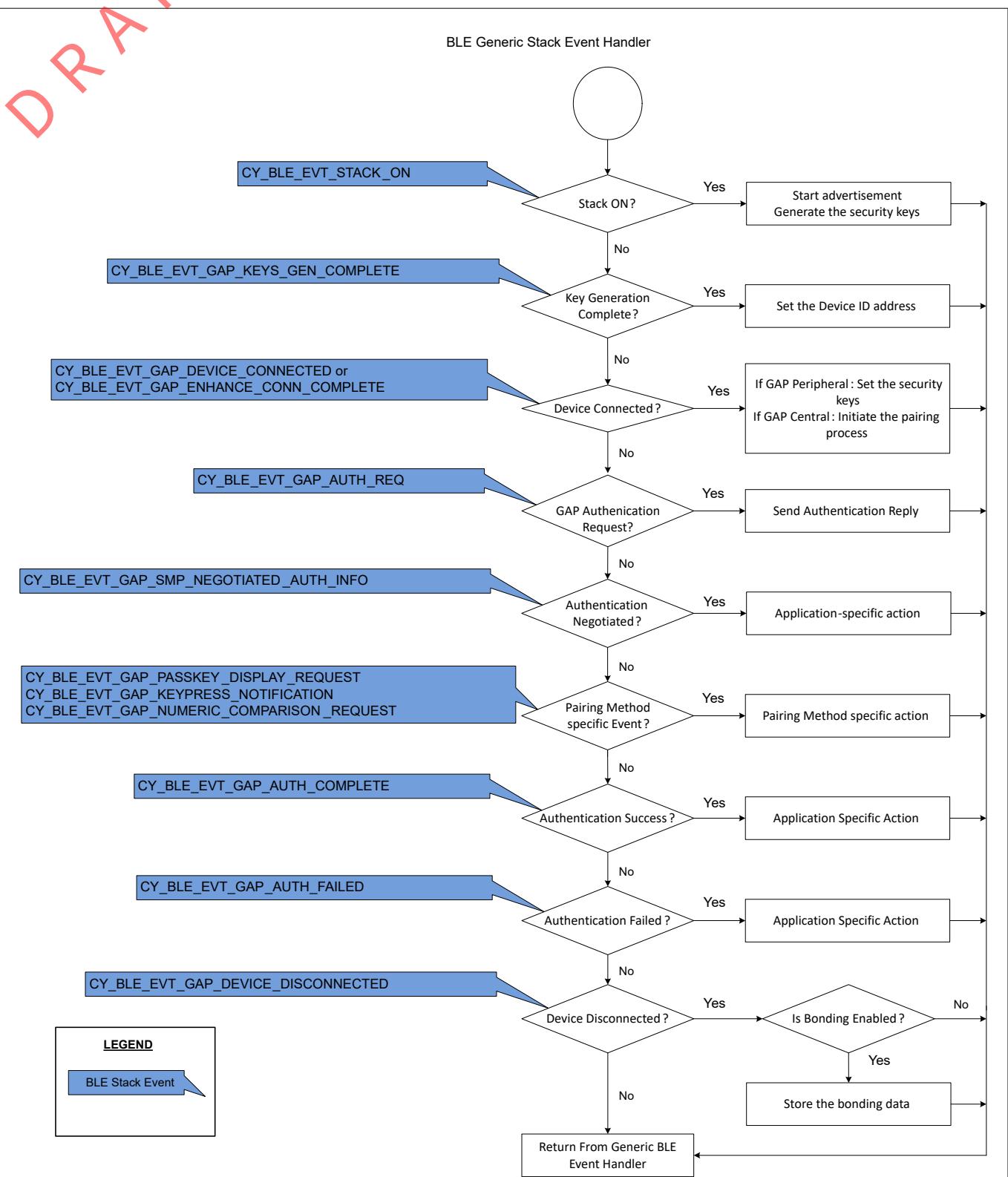


Figure 464 Firmware flow for secure BLE design with bonding

5 PSoC™ 6 application notes

~~DRAFT~~ Table 92 API functions for bonding

API function	Description
Cy_BLE_StoreBondingData()	This function writes the new bonding data from the RAM to the dedicated flash location as defined by the BLE middleware
Cy_BLE_GAP_GetBondList()	This function returns the count and list of bonded devices
Cy_BLE_GAP_RemoveDeviceFromBondList()	This function removes the specified device from the bond list

5.11.4.3 Additional BLE design considerations

Design considerations discussed in this section are optional steps that you can incorporate in your design based on your application requirements.

Link layer settings

The link layer (LL) is the part of the BLE protocol stack that handles advertising, scanning, creating, and maintaining connections. The LL of the BLE protocol stack supports BLE 4.2 features such as LE data packet length extension and link layer privacy.

LE data packet length extension

The LE data packet length extension feature enables applications to get higher throughput, lower power consumption, etc. These benefits are available under the following conditions:

- Both BLE devices support LE data packet length extension
- Higher-layer protocols use greater than the default (23 bytes) maximum transmission unit (MTU) size

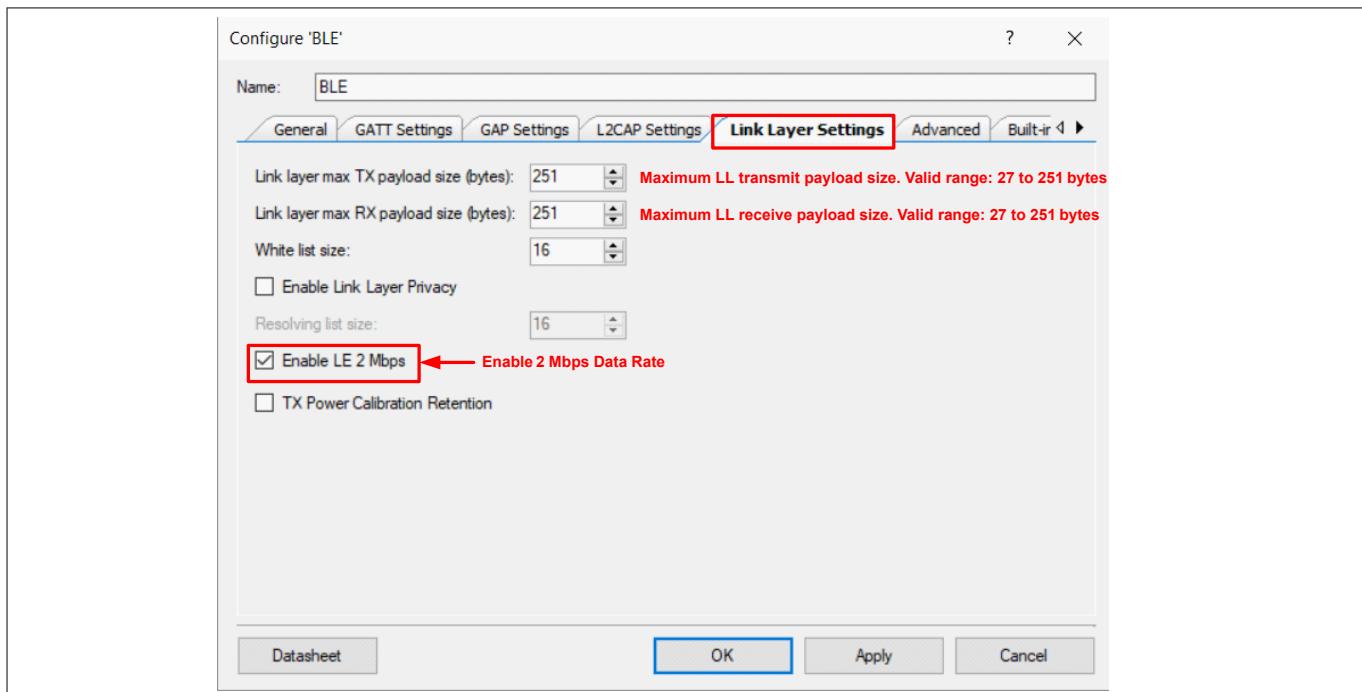


Figure 465 Configuring BLE link layer settings

When a BLE connection is established, the BLE stack automatically negotiates TX and the RX payload sizes with the peer device. Negotiated maximum TX and RX payload sizes are reported to the application through the CY_BLE_EVT_DATA_LENGTH_CHANGE BLE stack event.

~~5 PSoC™ 6 application notes~~

BLE stack event	Event handler action
CY_BLE_EVT_DATA_LENGTH_CHANGE	Informative event. This event reports the negotiated TX and RX payload sizes

~~LE 2-Mbps data rate~~

The BLE protocol stack in PSoC™ 6 BLE supports the BLE 5.0-compliant data rate of 2 Mbps. To use the 2-Mbps data rate, you need to enable this feature under the link layer settings tab of the BLE component as shown in [Figure 465](#). Note that the usage of the term PHY settings corresponds to the BLE data rate settings.

[Table 93](#) lists the BLE stack events and the action to be taken while configuring PHY settings.

Table 93 [BLE events associated with PHY configuration](#)

BLE stack event	Event handler action
CY_BLE_EVT_SET_PHY_COMPLETE	Indicates the completion of <code>Cy_BLE_SetPhy()</code>
CY_BLE_EVT_PHY_UPDATE_COMPLETE	Indicates that the controller has changed the transmitter PHY or receiver PHY in use
CY_BLE_EVT_GET_PHY_COMPLETE	Indicates the completion of <code>Cy_BLE_GetPhy()</code> .

[Table 94](#) lists the functions used for configuring the BLE data rate.

Table 94 [API functions for handling PHY settings](#)

API	Description
<code>Cy_BLE_SetPhy</code>	Allows the application to set the PHY for the current connection
<code>Cy_BLE_GetPhy</code>	Allows the application to read the current PHY setting for the specified connection

See the [BLE Component datasheet](#) for more details on LL configuration parameters. For a comprehensive list of BLE events and API functions, refer to Infineon BLE middleware library.

See [AN99209 – PSoC™ 4 BLE and PRoC BLE: Bluetooth® LE 4.2 Features](#) for an elaborate description on BLE 4.2 features. See [CE212742 – BLE 4.2 Data Length Security Privacy with PSoC™ 6 MCU with BLE Connectivity](#) for details on implementing LE secure connection and the data length extension (DLE) features of BLE.

~~DO NOT USE~~ 5 PSoC™ 6 application notes

5.11.5 BLE design examples

Now that you are familiar with basic firmware flow for BLE design using PSoC™ 6 BLE, it is time to get hands-on for designing BLE applications. This section discusses two BLE design examples illustrating the multi-master multi-slave (MMMS) capability of PSoC™ 6 BLE. Note that both the code examples discussed in this section are based on dual-CPU BLE architecture and the firmware is developed in an RTOS environment.

5.11.5.1 Multi-master multi-slave: Implementing four BLE slaves

This design illustrates the connectivity between PSoC™ 6 BLE, acting as a GAP peripheral and GATT server, and four BLE-enabled devices (a personal computer running the CySmart BLE host emulation tool, or a mobile device running the CySmart mobile app) acting as a GAP central and GATT client. This design is available as a code example: [CE223508 – PSoC™ 6 MCU implementing BLE multi-connection \(4 slaves\)](#).

5.11.5.1.1 About the design

In this design example, PSoC™ 6 MCU is configured as a GAP peripheral and a GATT server and can connect to as many as four GAP central devices.

PSoC™ 6 BLE implements the BLE multi-slave functionality that consists of the following services as shown in [Figure 466](#):

- Device information service
- Health thermometer service
- Custom service for RGB LED with color and intensity control
- 128-bit long characteristic read/write custom service
- A custom notification service

Connected GAP central devices can access the GATT database.

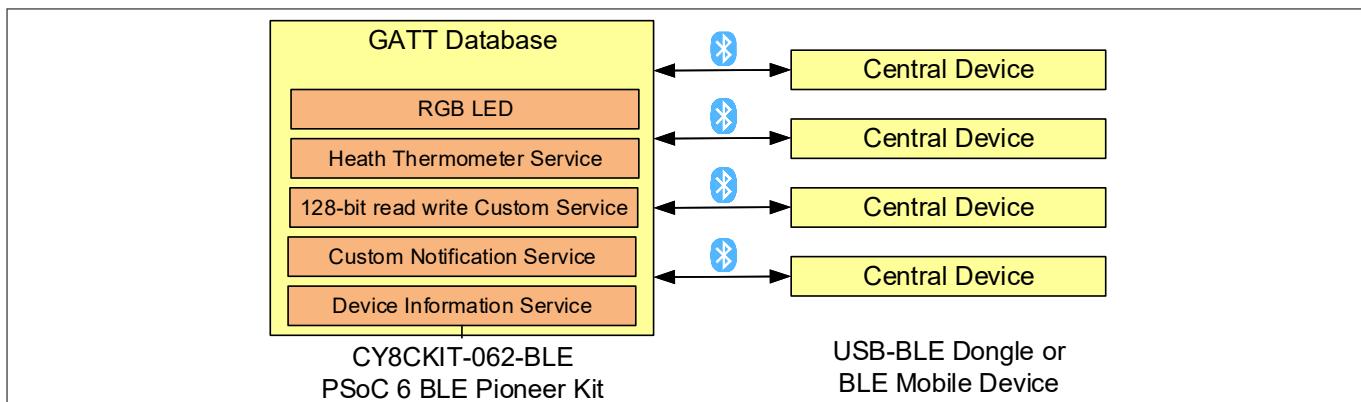


Figure 466 BLE services implemented by PSoC™ 6 BLE

The code example, [CE223508](#), features the following:

- BLE connectivity
 - Advertisement and connection with four GAP central devices
 - Five services (RGB LED, health thermometer service, device information, read/write 128-bit long custom service and custom notification service)
 - Data transfer over BLE using notifications, read, and write
- RGB LED color and intensity control using configurable digital blocks of PSoC™ 6 MCU
- The ADC in PSoC™ 6 MCU scans two differential channels and averages multiple samples without the need for CPU intervention for accurate temperature measurement from a thermistor circuit
- Device information service gives manufacturer and/or vendor information about the device

5 PSoC™ 6 application notes

- ~~DRAFT~~
- 128-bit read/write custom service
 - The custom notification sends notifications to connected devices about any changes to the GATT database. It sends a two-byte data: the first byte identifies the device that modified the data; the second byte identifies the characteristic that has been modified
 - Low-power operation using the deep sleep mode with multi-counter watchdog timer (MCWDT) and GPIO interrupts
 - The orange (LED8) and red (LED9) LEDs on the kit are used to show the status:
 - If the device is in hibernate mode, red LED will be ON
 - The orange LED will blink if the device is advertising

Figure 467 shows the firmware flow of [CE223508](#).

5 PSoC™ 6 application notes

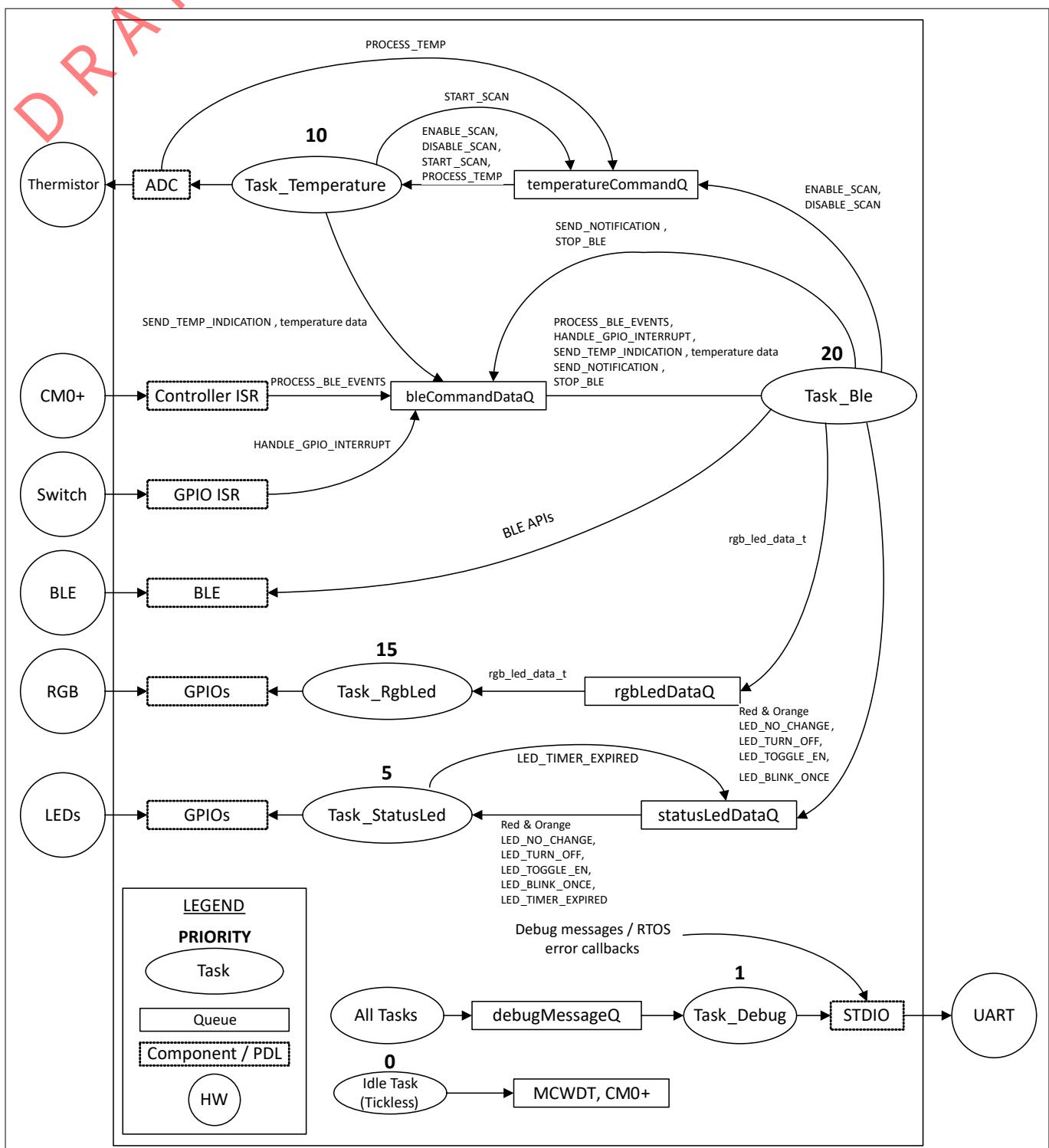


Figure 467

Firmware flow diagram for PSoC™ 6 MCU implementing BLE multi-connection (4 slaves)

5.11.5.2

Multi-master multi-slave: Implementing three BLE masters and one BLE slave

This design demonstrates how to configure PSoC™ 6 BLE in simultaneous multiple master and single slave mode of operation. This design configures PSoC™ 6 BLE as three BLE central roles and one BLE peripheral role (MMMS). The BLE multi-master single slave project is used in conjunction with the [CE215119](#) BLE battery level

~~5 PSoC™ 6 application notes~~

code example for PSoC™ 6 MCU or PSoC™ 4 devices to demonstrate the operation in simultaneous multiple master and single slave modes. For a detailed explanation on the design and implementation of the BLE design, see [CE224714 – PSoC™ 6 MCU Implementing BLE Multi-connection \(3 Masters 1 Slave\)](#).

~~5.11.5.2.1 About the design~~

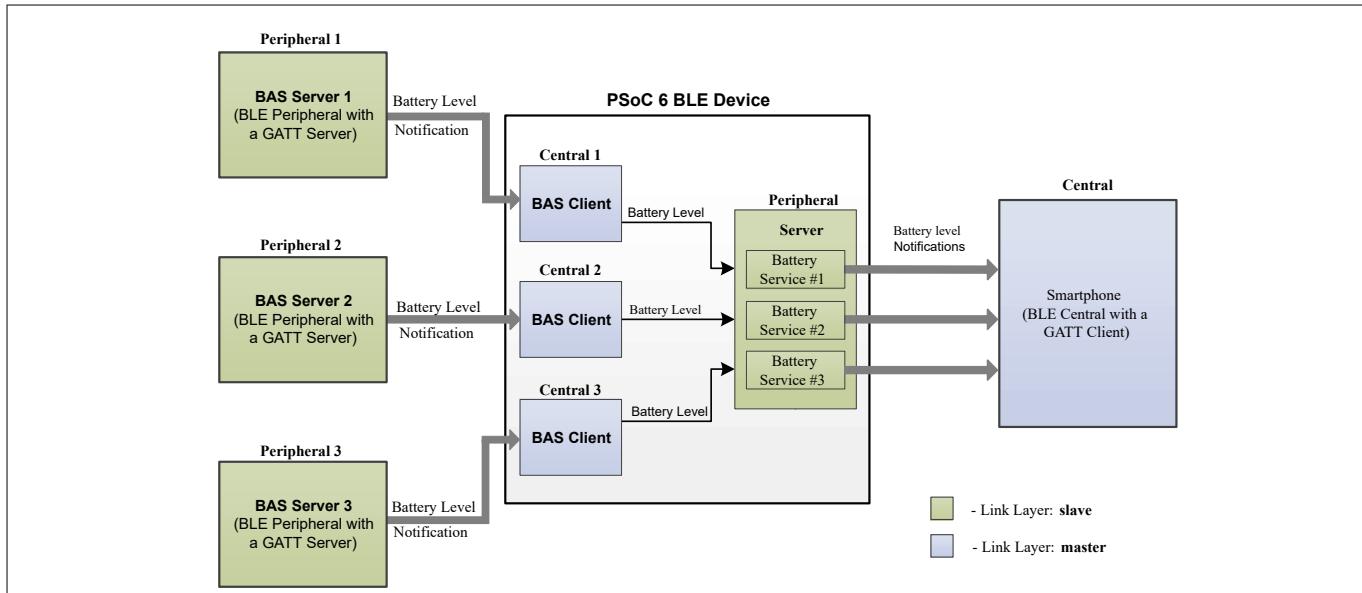


Figure 468 Multi-master single slave

The multi-master single slave project uses three BLE central connections and one peripheral connection:

- The central is configured as a generic attribute profile (GATT) client with a battery service that can communicate with a peer device in the generic access profile (GAP) peripheral and GATT server roles. Use the existing [CE215119 – BLE battery level code examples for PSoC™ 6 BLE/PSoC™ 4 devices](#) or an application that can simulate a GATT server with a battery service as a peer device.
- The peripheral is configured as a GATT server with three battery services. This configuration represents the battery level of the three peripherals that the device is connected to. [Figure 468](#) shows a block diagram of the multi-master single slave example

[Figure 469](#) shows the firmware flow of [CE224714](#).

5 PSoC™ 6 application notes

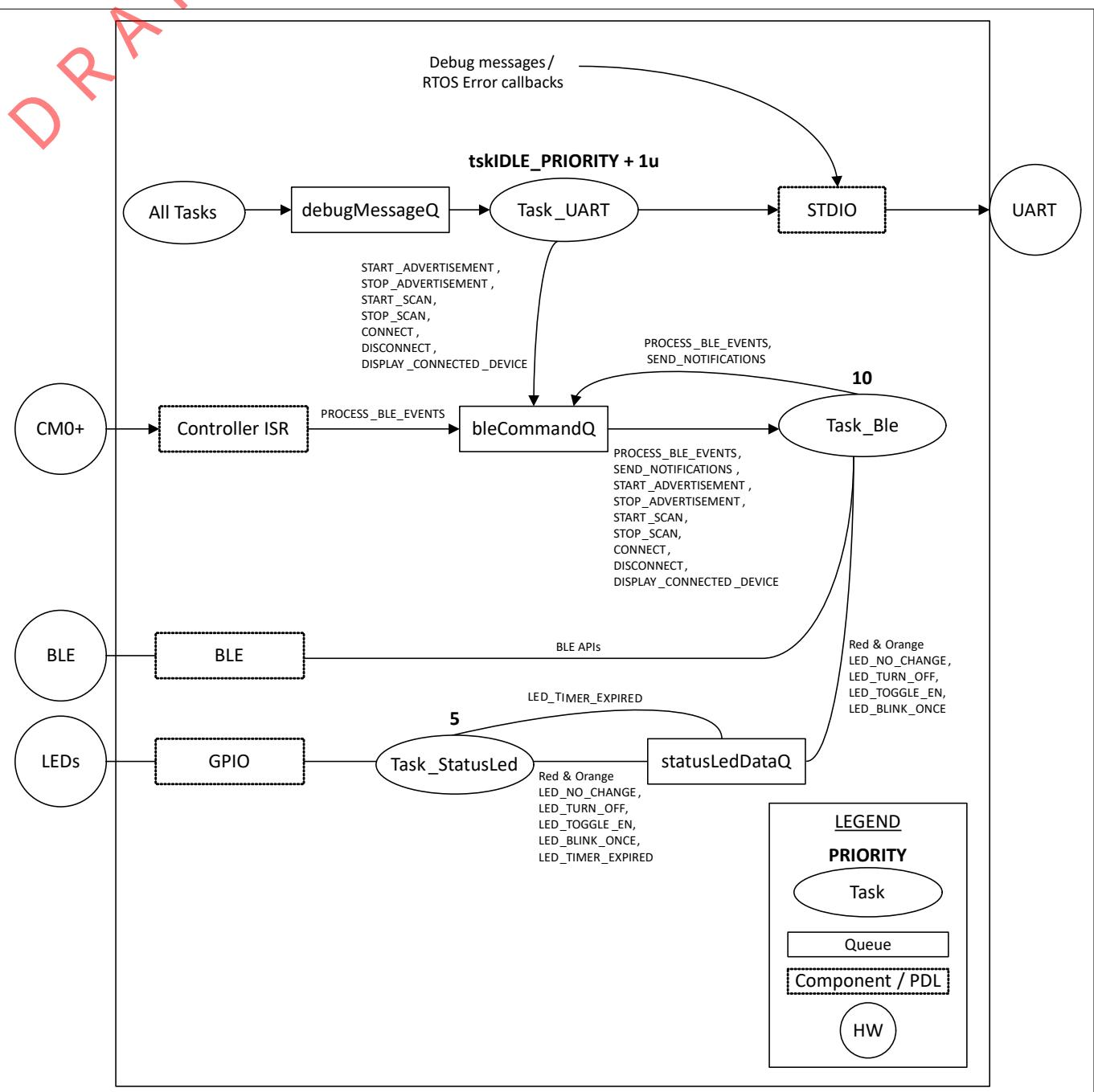


Figure 469 Firmware flow diagram for multi-master single slave

5 PSoC™ 6 application notes~~REDACTED~~
5.11.6 Summary

This application note introduced the BLE stack architecture and its implementation in PSoC™ 6 BLE. The application note further discussed the typical firmware flow and design considerations, such as low-power design and establishing BLE secure connection, for developing BLE applications. The code examples [CE223508](#) and [CE224714](#) discuss multi-master multi-slave (MMMS) feature of the BLE stack used with PSoC™ 6 BLE and illustrate the firmware flow for developing MMMS BLE applications.

~~5 PSoC™ 6 application notes~~

~~References~~

Application Notes

AN210781 – Getting Started with PSoC™ 6 MCU with Bluetooth® Low Energy (BLE) Connectivity	Describes PSoC™ 6 MCU with BLE connectivity devices and how to build your first PSoC™ creator project
AN218241 – PSoC™ 6 MCU Hardware Design Considerations	Describes how to design a hardware system around a PSoC™ 6 MCU device
AN215656 – PSoC™ 6 MCU: Dual-CPU System Design	Describes the dual-CPU architecture in PSoC™ 6 MCU, and shows how to build a simple dual-CPU design
AN221111 – PSoC™ 6 MCU: Creating a Secure System	Describes how to create a secure embedded system with PSoC™ 6 MCU
AN213924 – PSoC™ 6 MCU Bootloader Software Development Kit (SDK) Guide	Provides comprehensive information on how to use the bootloader software development kit (SDK) to develop bootloadable systems for PSoC™ 6 MCU products
AN219434 – Importing PSoC™ Creator Code into an IDE for a PSoC 6 MCU Project	Describes how to import the code generated by PSoC™ creator into your preferred IDE
AN99209 – PSoC 4 BLE and PRoC BLE: Bluetooth® LE 4.2 Features	Provides an overview of the Bluetooth® low energy (BLE) 4.2 features and explains their usage at the application level

PSoC™ creator code examples

CE223508 – PSoC™ 6 MCU Implementing BLE Multi-connection (4 Slaves)	Demonstrates the implementation of multi-slave functionality of the PSoC™ 6 MCU with BLE connectivity (PSoC™ 6 BLE) device
CE224714 – PSoC™ 6 MCU Implementing BLE Multi-connection (3 Masters 1 Slave)	Demonstrates how to configure the PSoC™ 6 MCU with Bluetooth® low energy (BLE) connectivity device in simultaneous multiple master and single slave modes of operation
CE212742 – BLE 4.2 Data Length Security Privacy with PSoC™ 6 MCU with BLE Connectivity	Demonstrates the new BLE 4.2 and 5.0 features of the PSoC™ creator BLE component
CE215119 – BLE Battery Level with PSoC™ 6 BLE	Demonstrates the operation of Bluetooth® low energy (BLE) battery service (BAS) using the PSoC™ creator BLE component
CE216767 – PSoC™ 6 MCU with Bluetooth® Low energy (BLE) Connectivity Bootloader	Demonstrates simple over the air (OTA) bootloading with PSoC™ 6 MCU with BLE connectivity

Device documentation

PSoC™ 6 MCU: PSoC™ 63 with BLE Datasheet	PSoC™ 6 MCU: PSoC™ 63 with BLE architecture technical reference manual
--	--

Development kit documentation

CY8CKIT-062-BLE PSoC™ 6 BLE Pioneer Kit

For more BLE code examples with PSoC™ 6 BLE, visit our GitHub repository: [PSoC™ 6 MCU BLE Connectivity Design s](#)

5 PSoC™ 6 application notes**5.11.7 Revision history**

Document version	Date of release	Description of changes
**	2018-09-21	New Application Note.
*A	2021-03-27	Migrated to Infineon template.
*B	2022-07-21	Template update.

5.12 AN217666 PSoC™ 6 MCU interrupts**About this document**

- .
- 1
- 2

Scope and purpose

This application note explains the interrupt architecture in PSoC™ 6 MCU and its configuration using PSoC™ Creator, ModusToolbox™, and the PSoC™ 6 Peripheral Driver Library (PDL) APIs. This document serves as a guide in developing projects that use interrupts. Advanced interrupt concepts such as interrupt latency, code optimization, and debug techniques are also explained.

To access an ever-growing list of hundreds of PSoC™ code examples, please visit our [code examples web page](#). You can also explore the Infineon video training library [here](#).

5 PSoC™ 6 application notes~~DO NOT USE~~
5.12.1 Introduction

An interrupt is a hardware signal or an event that transfers the execution of a program from the normal flow to an alternate set of instructions. An interrupt frees the CPU from continuously polling for a specific event, and only notifies and engages the CPU when the event occurs. The alternate program flow is referred to as an interrupt service routine or ISR. An ISR is also called an interrupt handler. After the interrupt is serviced, the program flow is reverted back to the flow that was interrupted. In system-on-chip (SoC) architectures such as PSoC™, interrupts are frequently used to communicate the status of on-chip peripherals to the CPU.

While interrupts refer to those events generated by peripherals external to the CPU such as timers, serial communication blocks, and port pin signals, an exception is an event generated by the CPU such as memory access faults and internal system timer events. PSoC™ 6 MCU supports interrupts and exceptions on both its Arm® Cortex®-M4 (CM4) and Cortex®-M0+ (CM0+) CPUs.

5.12.1.1 How to use this document

This document assumes that you are familiar with the PSoC™ 6 MCU architecture, and application development for PSoC™ devices using the Infineon PSoC™ Creator integrated design environment (IDE) and Peripheral Driver Library (PDL). For an introduction to PSoC™ 6 MCU, see [AN210781 - Getting Started with PSoC™ 6 MCU with Bluetooth® Low Energy \(BLE\) Connectivity](#). If you are new to PSoC™ Creator, ModusToolbox™, or PDL, see the [References](#) section for links to some of the available resources.

Note: Use [PSoC™ Creator version 4.2 or higher](#) for PSoC 6 MCU-based designs.

This document begins with a brief explanation of the PSoC™ 6 MCU interrupt architecture, with more details available in the [PSoC™ 6 MCU: PSoC™ 63 with BLE Architecture Technical Reference Manual \(TRM\)](#). To skip to an overview of writing firmware that uses interrupts, see [Configuring interrupts using PDL](#) or [Configuring interrupts using PSoC™ Creator](#) or [Configuring interrupts using ModusToolbox™](#) sections respectively. Code examples that show how to use interrupts for various peripherals are listed in the [References](#) section.

The [Debugging tips](#) section provides a few tips on finding and resolving common issues encountered while using interrupts. More complex topics are covered in [Advanced interrupt topics](#).

~~5 PSoC™ 6 application notes~~

~~5.12.2 PSoC™ 6 MCU interrupt architecture~~

PSoC™ 6 MCU contains two CPUs: CM4 and CM0+. Interrupt signals to each CPU are handled by the respective Nested Vectored Interrupt Controller (NVIC). The NVIC enables/disables any interrupt based on the user configuration. It also resolves the interrupt priority when multiple requests occur at the same time and supports nested interrupts to allow a higher-priority interrupt to be serviced before a lower-priority ISR.

PSoC™ 6 MCU also supports a wakeup interrupt controller (WIC) and multiple synchronization blocks. The WIC block allows the CPU to wake up from sleep or deep sleep low-power modes using interrupts. The WIC block remains active while the NVIC, processor core, and other device peripherals shut down. When an interrupt triggers, the WIC activates the power management system, which restores the NVIC and the processor core along with other peripherals. Each CPU has independent WIC settings.

Natively, CM4 supports up to 240 interrupts, while CM0+ supports 32 interrupts. The number of CPU interrupts available to the user varies depending on the device, see [Table 95](#).

CM4 supports configurable interrupt priority from 0 to 7. CM0+ supports priority from 0 to 3.

There are up to 175 interrupt sources (also referred to as system interrupts) in a PSoC™ 6 MCU device. System interrupts can trigger either or both CPUs.

The WIC block can wake up a CPU from deep sleep power mode. [Table 96](#) lists the interrupt sources that can wake a CPU from deep sleep.

One or more system interrupts can be selected as the source for the CPU non-maskable interrupt (NMI), see [Table 95](#).

Table 95 **Interrupt features in PSoC™ 6 MCU**

Parameter	CY8C61x6/7, CY8C62x6/7, CY8C63xx	CY8C62x8/A	CY8C62x5	CY8C62x4
Number of system interrupts ("N")	147	168	174	175
Number of deep sleep-capable system interrupt sources ("W")	41	39	39	45
Number of CM0+ interrupt vectors available	32 (8 deep sleep-capable)	8 hardware (deep sleep-capable) 8 software triggered	8 hardware (Deep sleep-capable) 8 software triggered	8 hardware (Deep sleep-capable) 8 software triggered
Number of system interrupts that can be connected to a CM0+ multiplexer/vector	1	All (168)	All (174)	All (175)
Number of CM4 interrupt vectors available	240	240	240	240
Number of system interrupts that can be connected to a CM4 multiplexer/vector	1 (1:1 mapping)	1 (1:1 mapping)	1 (1:1 mapping)	1 (1:1 mapping)
Number of system interrupts that can be connected to CM0+/CM4 NMI interrupt	1	4	4	4

5 PSoC™ 6 application notes

5.12.2.1 CY8C61x6/7, CY8C62x6/7, and CY8C63xx interrupt architecture

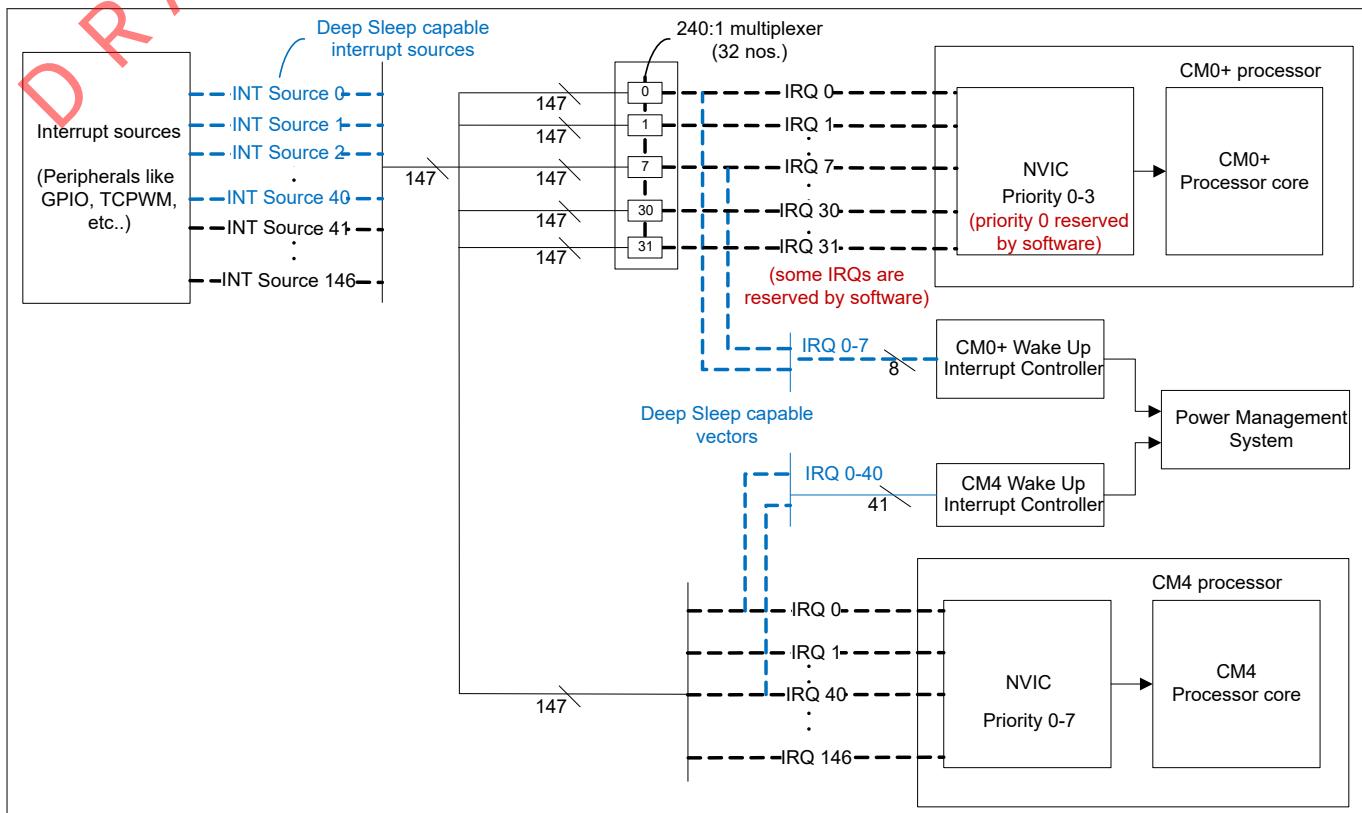


Figure 470 CY8C61x6/7, CY8C62x6/7, and CY8C63xx interrupt architecture

CY8C61x6/7, CY8C62x6/7, and CY8C63xx devices support up to 147 system and peripherals interrupt sources. For CM4, the 147 interrupt sources are directly mapped to its first 147 IRQ lines, i.e., INT source n is connected to IRQ n, where 'n' = 0 to 146. For CM0+, a 240:1 multiplexer is present in front of each of 32 IRQs and redirects any of the 147 interrupts to one of CM0+ IRQ lines. This enables any interrupt source to trigger any CM0+ IRQ.

5 PSoC™ 6 application notes

5.12.2.2 CY8C62x4, CY8C62x5, and CY8C62x8/A interrupt architecture

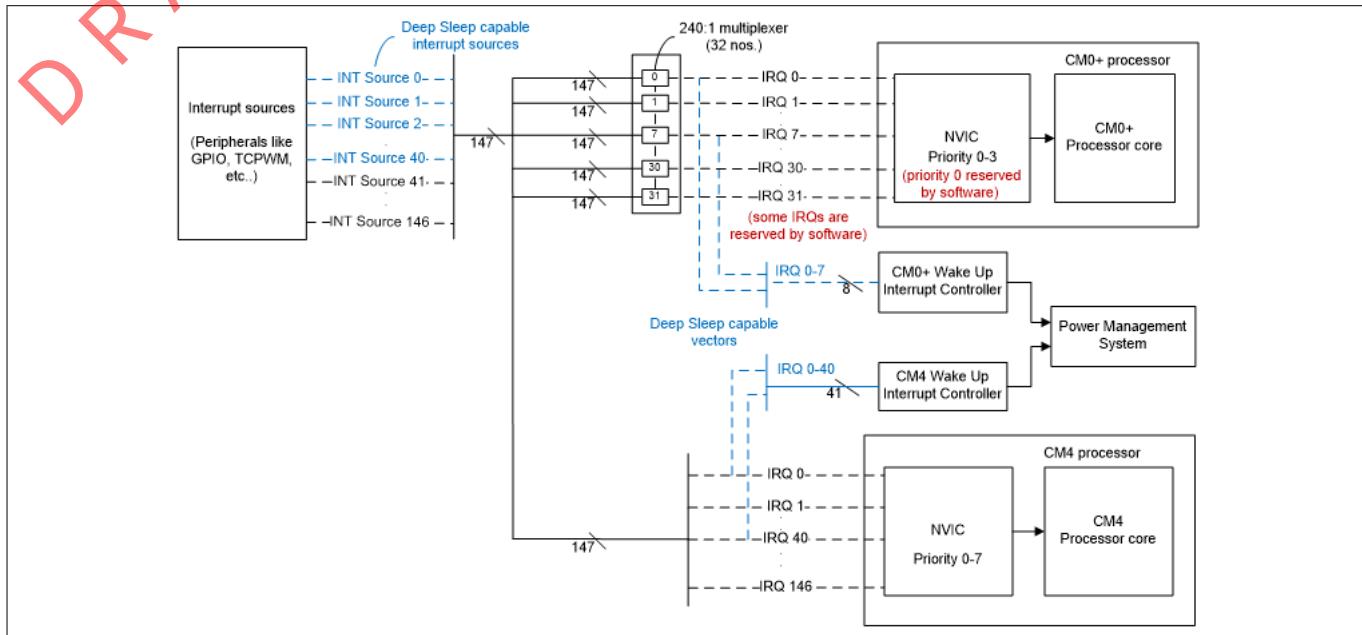


Figure 471 CY8C62x4, CY8C62x5, and CY8C62x8/A interrupt architecture

In CY8C62x4, CY8C62x5, and CY8C62x8/A devices, the ‘N’ interrupt sources are directly mapped to the first ‘N’ IRQ lines of the CM4. The CM0+ supports 16 interrupts, of which the first 8 interrupts (IRQ 0 – 7) can be triggered by a peripheral interrupt source; the other 8 are software-triggered interrupts. One or more system interrupts (upto ‘N’) can be assigned as the interrupt source for each of the IRQ 0 – 7 lines. This allows multiple interrupt sources to be connected to the same CPU interrupt simultaneously.

The WIC block supports up to ‘W’ interrupts that can wake up a CPU from deep sleep power mode; see “Number of deep sleep-capable system interrupt sources” in [Table 95](#).

[Table 96](#) lists the interrupt sources that can wake a CPU from deep sleep.

Note: When using Infineon software (PDL or PSoC™ Creator), certain software restrictions apply on the number of CPU interrupts available to user and interrupt priorities. See [Configuring interrupts using PDL](#) for details.

5.12.2.3 Types of interrupts

There are two kinds of interrupt sources in PSoC™ 6 MCU:

- Fixed-function interrupt sources

These are predefined interrupt sources from on-chip peripherals such as GPIO, TCPWM, SCB, and BLE Radio. Interrupts from fixed-function sources are generated from configurable events; for example, an interrupt on a rising edge for example signal on an input pin (GPIO), or an interrupt on a counter overflow (TCPWM).

- Universal Digital Block (UDB) interrupt sources

UDBs consist of programmable logic devices (PLDs), datapaths, and flexible routing, which can be used to synthesize different digital functions such as Timer, PWM, UART, SPI and many more. In contrast to fixed-function interrupt sources, any digital signal generated in a UDB can trigger an interrupt. The signals are routed to the interrupt controller through the routing fabric known as Digital System Interconnect (DSI). UDB sources are available only in CY8C61x6/7, CY8C62x6/7, and CY8C63xx devices

For a complete list of interrupt sources in PSoC™ 6 MCU, see [Appendix A](#).

~~5 PSoC™ 6 application notes~~

~~5.12.2.3.1 Level and pulse interrupts~~

Both CM0+ and CM4 NVICs support level and pulse signals on IRQ lines. The classification of an interrupt as level or pulse is based on the interrupt source. A fixed-function interrupt is treated as level-sensitive. For the DSI sources, which include the UDB, the interrupt can be configured as either rising-edge-triggered or level-triggered. This configuration is available only in PSoC™ Creator. For more details on selecting the interrupt type, refer to the PSoC™ Creator Component datasheet or PDL API reference for the interrupt source.

For level interrupts, if the interrupt signal is still HIGH after completing the ISR, the interrupt is still pending and the ISR is executed again. [Figure 472](#) illustrates the timing diagram for level-triggered interrupts, where the ISR is executed if the interrupt signal is HIGH.

For pulse interrupts, while the ISR is being executed by the CPU, one or more rising edges of the interrupt signal are logged as a single pending request. The pending interrupt is serviced again after the current ISR execution is complete. [Figure 473](#) illustrates the timing diagram for pulse interrupts.

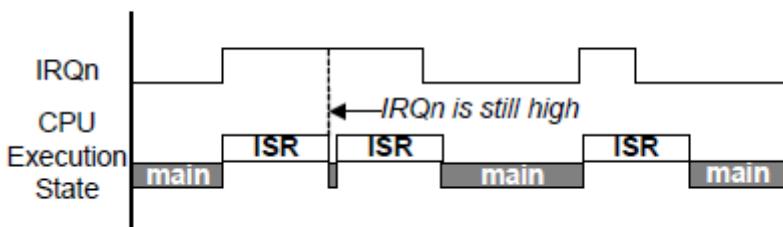


Figure 472 Level interrupts

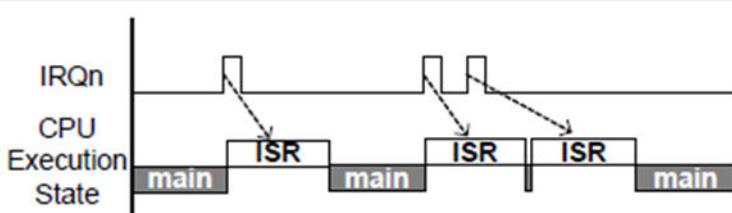


Figure 473 Pulse interrupts

Note: The GPIO interrupt logic has additional circuitry to support interrupts on the rising edge, falling edge, and both edges. See the I/O System chapter in [PSoC™ 6 MCU Architecture TRM](#) for more information.

5.12.2.4 Interrupts and Power modes

PSoC™ 6 MCU has the following system Power modes: Low-Power (LP), Ultra-Low-Power (ULP), deep sleep, and hibernate. The Arm® CPU Power modes are active, sleep, and deep sleep; these are available in system LP and ULP Power modes.

In CPU active modes, CPUs execute code; all memory blocks and peripherals are available.

In all other Power modes (sleep, deep sleep, hibernate), CPU clocks are turned off and code execution is halted.

All peripherals available in active modes are also available in the sleep, deep sleep, and hibernate modes. Any peripheral interrupt, masked to the CPU, wakes up the CPU to Active mode.

Only a subset of peripherals operate in deep sleep mode. Interrupts from these peripherals cause a CPU to wake up to active mode. [Table 96](#) lists these peripherals. Each CPU has a Wakeup Interrupt Controller (WIC) to wake up the CPU from its deep sleep mode. Deep sleep wakeup functionality is supported only on the first 8 IRQs (0 to 7) on CM0+ and first 'W' IRQs on CM4, see [Table 95](#).

~~5 PSoC™ 6 application notes~~

During hibernate mode, all peripherals and clocks are turned off and only certain sources like Low Power Comparator, RTC, a dedicated WAKEUP pin, or an XRES event can wake up the device. The wakeup action is a device reset instead of an interrupt to the CPU.

For more details on device Power modes CPU sleep and wakeup behavior due to interrupts, see [AN219528 – PSoC™ 6 MCU Low-Power Modes and Power Reduction Techniques](#) or [PSoC™ 6 MCU Architecture TRM](#).

Table 96 List of Deep Sleep Wakeup-Capable Interrupts

Interrupt Source	Interrupt Source Number			
	CY8C63xx	CY8C62x6/7CY 8C61x6/7	CY8C62x8/ ACY8C62x5	CY8C62x4
GPIO Port Interrupt	0–14	0–14	0–14	0–14
GPIO All Ports	15	15	15	15
GPIO Supply Detect Interrupt	16	16	16	16
Low Power Comparator Interrupt	17	17	17	17
Serial Communication Block Interrupt	18	18	18	18
Multi Counter Watchdog Timer	19, 20	19, 20	19, 20	19, 20
Backup Domain Interrupt	21	21	21	21
Other combined Interrupts for SRSS	22	22	22	22
Combined Continuous Time Block (CTBm) Interrupt	23	23	–	–
Bluetooth® Radio Interrupt	24	–	–	–
Inter Process Communication Interrupt	25–40	25–40	23–38	23–38
SAR ADC Interrupt	–	–	–	39, 40
Individual Continuous Time Block (CTBm) Interrupt	–	–	–	41
PASS Timer interrupt	–	–	–	42
PASS FIFO Interrupt	–	–	–	43, 44

5.12.2.5 CPU sleep and wakeup

There are two instructions that can cause the CPU to enter its sleep modes: the “Wait-for-Interrupt” [`__WFI()`] and “Wait-for-Event” [`__WFE()`]. When a WFI instruction is executed, the CPU enters sleep or deep sleep (depending on the SLEEPDEEP bit of the SCR register) and wakes up on an interrupt request (with a higher priority than the current priority level) or on debug requests. The WFE instruction is like WFI but wakes up on the next interrupt or on events like Send Event (SEV instruction), external event, or debug signals. See [AN219528](#) for more details on sleep and wakeup instructions.

Normally, when an ISR is done executing, CPU execution returns to where it was before the ISR. PSoC™ 6 MCU supports the “Sleep-on-Exit” feature where the CPU enters or returns to sleep or deep sleep (a state similar to WFI) as soon as it completes ISR execution. As seen in [Figure 474](#), when this feature is enabled, only one WFI instruction is needed to enter a sleep mode; the CPU returns to sleep after each ISR instead of the execution returning to main. The Sleep-on-Exit feature reduces the active cycles of the CPU and reduces the energy consumed by the stacking (PUSH to stack) and unstacking (POP from stack) of processes between interrupts. [Nested interrupts](#) are also supported when Sleep-on-Exit is enabled.

5 PSoC™ 6 application notes

The Sleep-on-Exit feature is enabled by setting SLEEPONEXIT bit of the SCR register. There is also a PDL function available – Cy_SysPm_SleepOnExit; see [Configuring interrupts using PDL](#) for details.

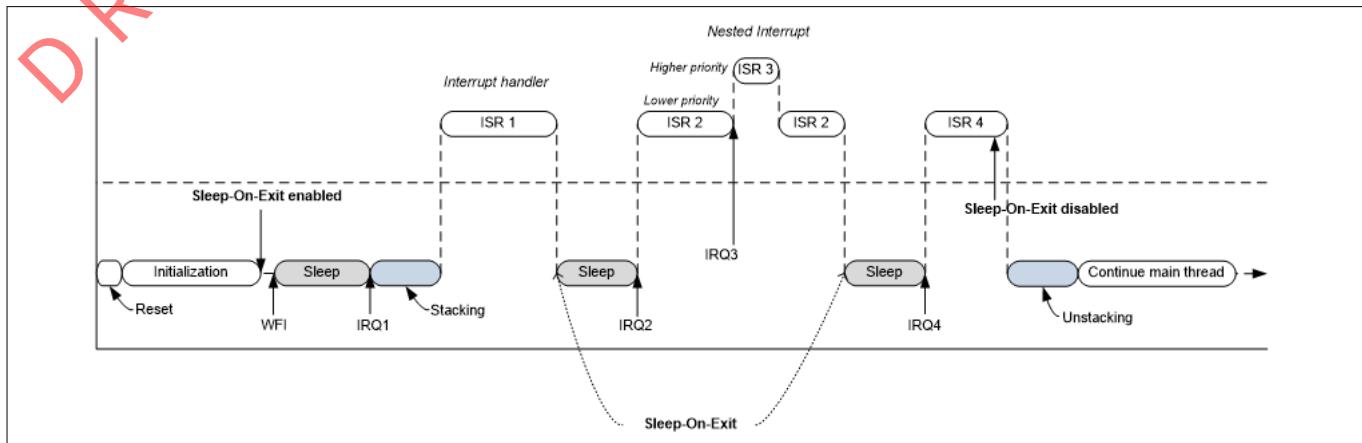


Figure 474 **Sleep-on-Exit function**

5 PSoC™ 6 application notes

~~DRAFT~~ 5.12.3 Interrupt configuration

This section lists the steps needed to set up interrupts on a PSoC™ 6 MCU device, without going into details of the software used to do them. These steps are common to both CM0+ and CM4 unless specified otherwise, and must be done for each CPU separately.

- Out of device reset, all interrupts are disabled, and interrupt priorities are set to zero
- Configure the priority level of the required IRQ in the NVIC
- Configure the interrupt path
- Choose which interrupt source is connected to the desired IRQ of the CPU. For CM0+, select the appropriate peripheral interrupt to be connected to the CPU. For CM4, this is not configurable. Interrupt source n is always connected to IRQn
- Configure the interrupt source (peripheral) and enable its interrupt
- Configure the vector table with the address of the ISR (vector). The vector table stores the entry addresses for each exception handler; see [Exception Vector Table in Interrupts chapter of PSoC™ 6 MCU Architecture TRM](#).
- Optional: Clear pending interrupt states in the NVIC
- If enabling a previously disabled interrupt, it is a good practice to clear the pending state of the NVIC before enabling the interrupt. This prevents any false trigger caused by previous interrupts that created a pending state
- Enable the interrupt in the NVIC
- Enable global interrupts. Interrupt configuration is complete

An enabled interrupt is triggered when the hardware signal from the interrupt source is active and there is no higher priority interrupt that is executing. When this happens, CPU execution jumps to the location in its vector table that corresponds to the triggered interrupt. This location contains the address of the ISR associated with that interrupt.

The ISR executes the tasks required to handle the interrupt. Typically, the first thing an ISR does is clearing the interrupt source to avoid re-entering the ISR. When the ISR terminates, the CPU returns to the address it was executing before it was interrupted. The following sections describe the software tools available for performing the steps described.

5.12.3.1 Configuring interrupts using ModusToolbox™

ModusToolbox™ applications support both the PSoC™ 6 Hardware Abstraction Layer (HAL) and Peripheral Driver Library (PDL) libraries.

5.12.3.1.1 Using HAL

The HAL gives an abstracted interface to configure and use various blocks on Infineon MCUs. There is no separate block for interrupts in HAL. The interrupts for different blocks are configured using the HAL APIs specific to those blocks.

For example, in the case of a GPIO Interrupt, interrupts arising from the GPIO block are configured using the GPIO HAL APIs. The steps to configure the GPIO HAL block for this example include:

- Initializing the GPIO pin: The GPIO pin, direction, drive mode, and initial value are passed to the `cyhal_gpio_init` HAL API function
- Registering the interrupt callback function: The callback function for the interrupt is registered with the GPIO pin using `cyhal_gpio_register_callback`
- Configuring the interrupt: The interrupt settings such as the GPIO event and interrupt priority are configured using `cyhal_gpio_enable_event`
- See the [Interrupts on GPIO events code snippet](#) in GPIO HAL API Reference Guide

5 PSoC™ 6 application notes

For more details on HAL APIs, see [PSoC™ 6 HAL](#) on GitHub.

5.12.3.1.2 Using Device Configurator and PDL

Interrupts can be configured in ModusToolbox™ using the [Device Configurator](#), the GUI-based tool used to enable and configure MCU peripherals and their interrupt parameters.

[Figure 475](#) shows the configuration of a GPIO pin to generate interrupt on a falling edge.

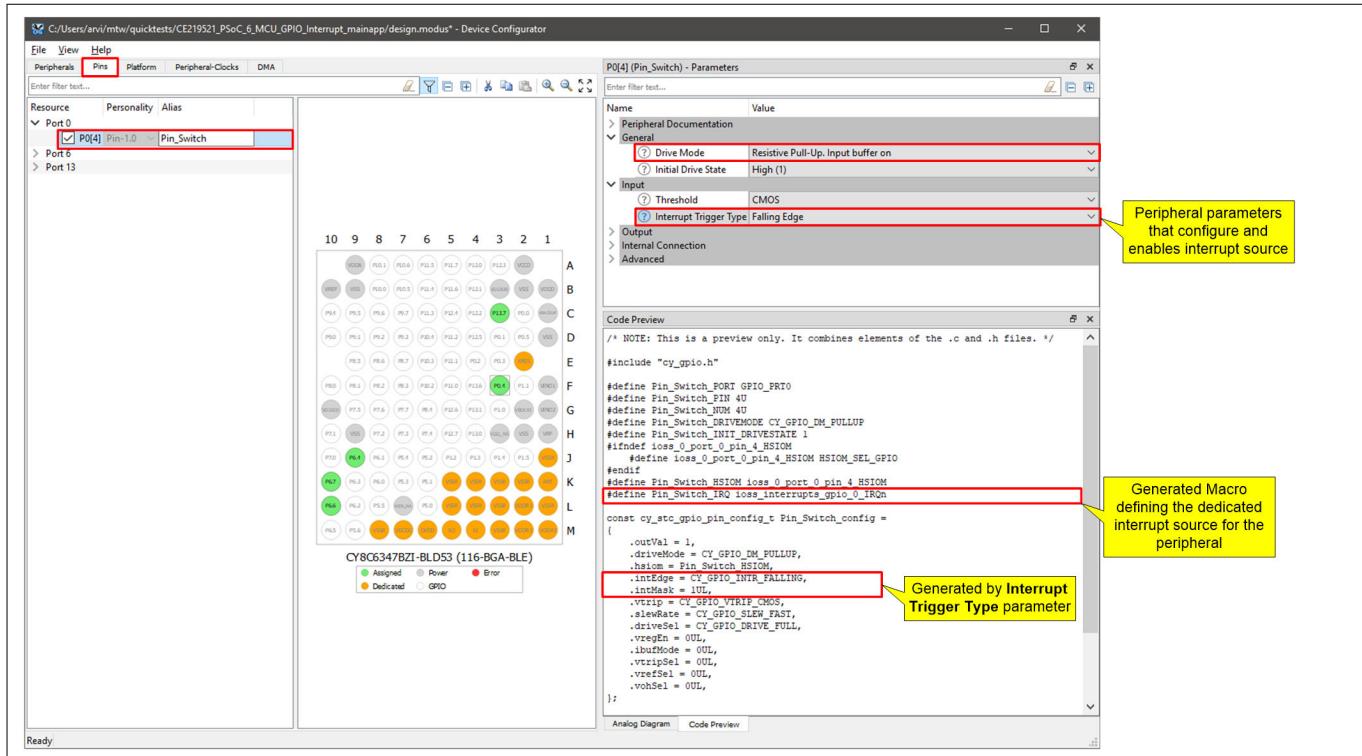


Figure 475 ModusToolbox™ peripheral configuration

Based on the configuration, ModusToolbox™ generates the ‘C’ code to achieve the desired configuration. The code generated can be viewed in the Code Preview pane; it is added to relevant cycfg_xxx.c/h files in the TARGET_<BSP name>/COMPONENT_BSP_DESIGN_MODUS/GeneratedSource folder in the ModusToolbox™ application. The generated code includes macros defining the interrupt source numbers and any peripheral configuration that is necessary to set up and enable the interrupt source. This simplifies the process of searching for the dedicated interrupt numbers in the device header file. The user application only needs to enable the interrupt vector on the CPU and assign an interrupt handler function as described in [Configuring interrupts using PDL](#).

5.12.3.2 Configuring interrupts using PDL

The Peripheral Driver Library (PDL) simplifies software development for the PSoC™ 6 MCU architecture. The PDL reduces the need to understand register usage and bit structures, so easing software development for the extensive set of peripherals available.

Note: The ModusToolbox™ software version of PDL is available at the [Infineon GitHub site](#). It is not compatible with PSoC™ Creator. The ModusToolbox™ version of the PDL includes support for new PSoC™ 6 MCU devices and drivers. It also supports macOS and Linux hosts, as well as Windows. Developers should move to the ModusToolbox™ package as projects and schedules permit. PDL v3.1 is designed for and works with PSoC™ Creator. PDL v3.1 is expected to be the final PSoC™ Creator-compatible release. PDL v3.0.x is installed along with the PSoC™ Creator 4.2 development tools.

~~5 PSoC™ 6 application notes~~

PDL API function calls are used to configure, initialize, enable, and use a peripheral driver. One such driver is System Interrupts (SysInt). SysInt provides structures and functions to configure and enable interrupt functionality. PDL also supports the CMSIS-Core libraries which include **NVIC functions** used for interrupt configuration.

The following steps use PDL and NVIC APIs to set up an interrupt to trigger on a signal from a peripheral.

- Configure the peripheral to generate the interrupt. For example, for a GPIO, configure the drive mode (pull up or pull down), interrupt signal generation on falling or rising edge, and unmask the interrupt. Refer to the PDL API reference documentation for your peripheral for this information
- Configure the interrupt using the structure provided by the SysInt API.

The structure is defined in the PDL SysInt driver file `cy_sysint.h`:

```
* Initialization configuration structure for a single interrupt channel */
typedef struct {
    IRQn_Type      intrSrc;    /*< Interrupt source */
    #if (CY_CPU_CORTEX_M0P)
        cy_en_intr_t   cm0pSrc;   /*< (CM0+ only) Maps cm0pSrc device
                                    interrupts to intrSrc */
    #endif
    uint32_t       intrPriority; /*< Interrupt priority number (Refer to
                                __NVIC_PRIO_BITS) */
} cy_stc_sysint_t;
```

This structure is used to configure the following (see [Figure 476](#) for a quick summary):

- Interrupt Source (intrSrc)
 - These are the dedicated interrupt numbers as defined in the device header file (example: `cy8c6247bzi_d44.h`)
 - This selection depends on which CPU you want to assign the interrupt to
 - For CM4, this number represents both the interrupt number of the source as well as the CPU IRQ number. Select the interrupt number of the peripheral interrupt you wish to route to the CPU. For example, to route Port 0 GPIO interrupt, assign a value of `ioss_interrupts_gpio_0 IRQn (=0)`.
 - For CM0+, this number represents one of the 32 multiplexers available for routing an interrupt to CM0+. Because each multiplexer is connected to a dedicated CM0+ IRQ line, use this to select the target CM0+ IRQ number. For example, to use multiplexer #4 (CM0+ IRQ#4), use “`NvicMux4 IRQn (=4)`”.
- CM0+ interrupt number (cm0pSrc)
 - This parameter is applicable only for CM0+.
 - This represents the interrupt number of the source, which is to be routed to the multiplexer/CM0+ interrupt generator logic, selected using the intrSrc parameter. Select the interrupt number of the peripheral interrupt you wish to route to the CPU; for example, to route Port 0 GPIO interrupt, assign a value of “`ioss_interrupts_gpio_0 IRQn (=0)`”.
- Interrupt priority (intrPriority)
 - Set the priority of the interrupt. For CM4, supported priorities are 0 to 7. For CM0+, supported priorities are 0 to 3

Notes:

- On CM0+, some IRQs are reserved for use by software and not available to the user. See “[Configuration Considerations](#)” under [SysInt driver in PDL API reference documentation](#) for the list of reserved IRQs
- On CM0+, the interrupt priority 0 is reserved for system calls

5 PSoC™ 6 application notes

DRAFT

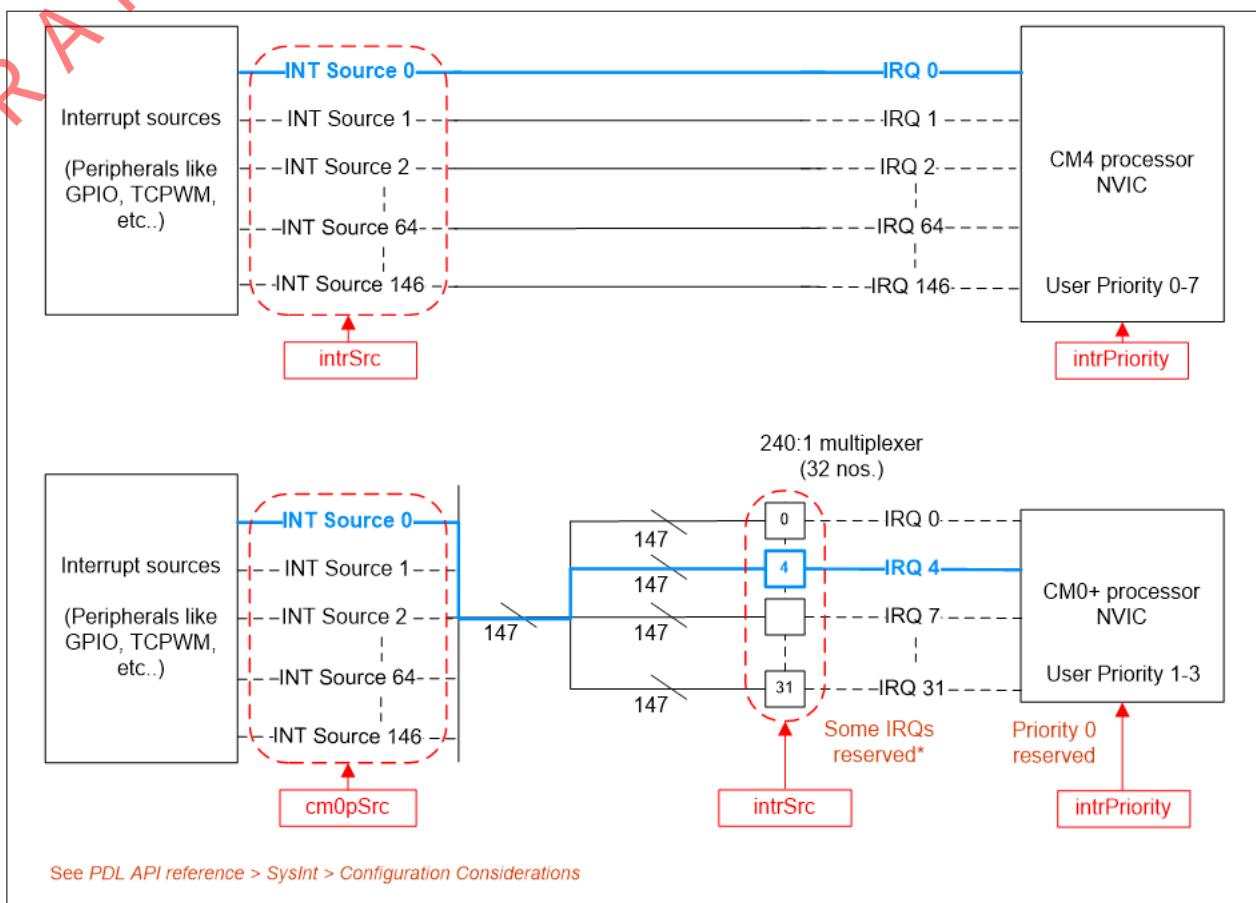
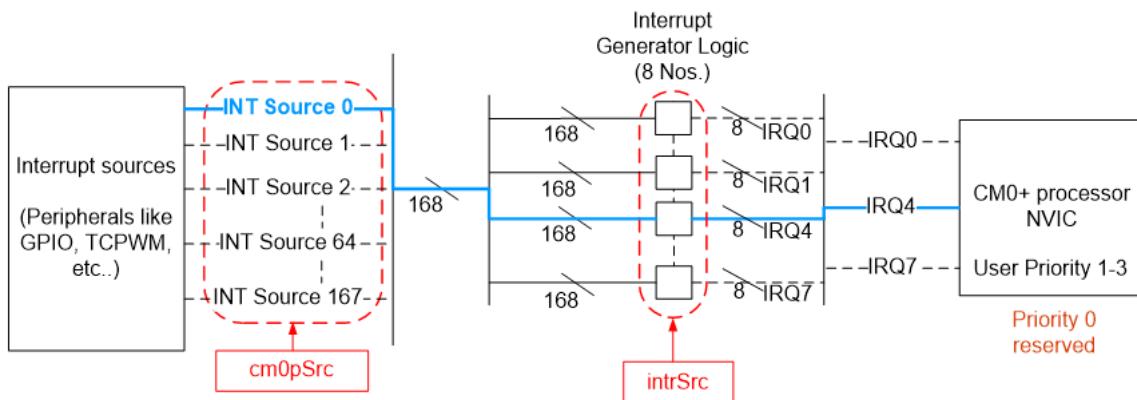


Figure 476 **SysInt PDL structure parameters (highlighted in red) used for interrupt configuration (sample configured path highlighted in blue)**

For CY8C62x4, CY8C62x5 and CY8C62x8/A devices:



- Call `Cy_SysInt_Init(&SysInt_SW_cfg_1, ISR_1_handler)`.
- Here, `SysInt_SW_cfg_1` is the name of the configured structure. `ISR_1_handler` is the name of the interrupt handler that executes when the interrupt triggers. This function applies the routing and priority configuration of the interrupt but does not enable it
- Call `NVIC_ClearPendingIRQ(SysInt_SW_cfg_1.intrSrc)` to clear any pending interrupts

5 PSoC™ 6 application notes

- ~~DRAFT~~
- Call `NVIC_EnableIRQ(SysInt_SW_cfg_1.intrSrc)` to enable the interrupt
 - Call the `_enable_irq()` function to enable global interrupts. This is safe to perform as the first step, as individual CPU interrupts have not been enabled yet. You can also perform this later but interrupts are disabled at startup unless this is called

In addition to the PDL `SysInt` driver, the system power modes (`SysPm`) driver API enables the Sleep-on-Exit feature. If sleep or deep sleep mode is used in the application along with interrupts, this feature enables the firmware to keep the system in a sleep mode almost all the time, only wake up to execute the interrupt and then immediately go back to the same sleep mode. The program does not return to the main function and stays either in the interrupt handler or in the same sleep state unless the Sleep-on-Exit feature is disabled again.

```
Cy_SysPm_SleepOnExit(true);
```

5.12.3.3 Configuring interrupts using PSoC™ Creator

PSoC™ Creator provides a graphical interface for routing signals from peripherals to a CPU IRQ line. PSoC™ Creator provides an Interrupt (`SysInt`) Component. This component is a UI element on top of the `SysInt` PDL driver discussed in the previous section. Based on the configuration in the Component, PSoC™ Creator generates code to initialize peripherals, route interrupts, and populate the interrupt configuration structure. This reduces the amount of code you must write when setting up interrupts.

The following section shows steps to use PSoC™ Creator to configure an interrupt. See the [References](#) section for code examples.

5.12.3.3.1 Using the schematic (TopDesign)

Drag and drop a Component from the Component Catalog onto the TopDesign. Use TopDesign to place and configure peripherals that provide a source of interrupt. Consult the Component datasheet for information on the peripheral's interrupt configuration. Some peripherals provide an interrupt terminal (e.g., TCPWM). Place an instance of the `SysInt` Component and connect it to the interrupt terminal of the peripheral.

5 PSoC™ 6 application notes

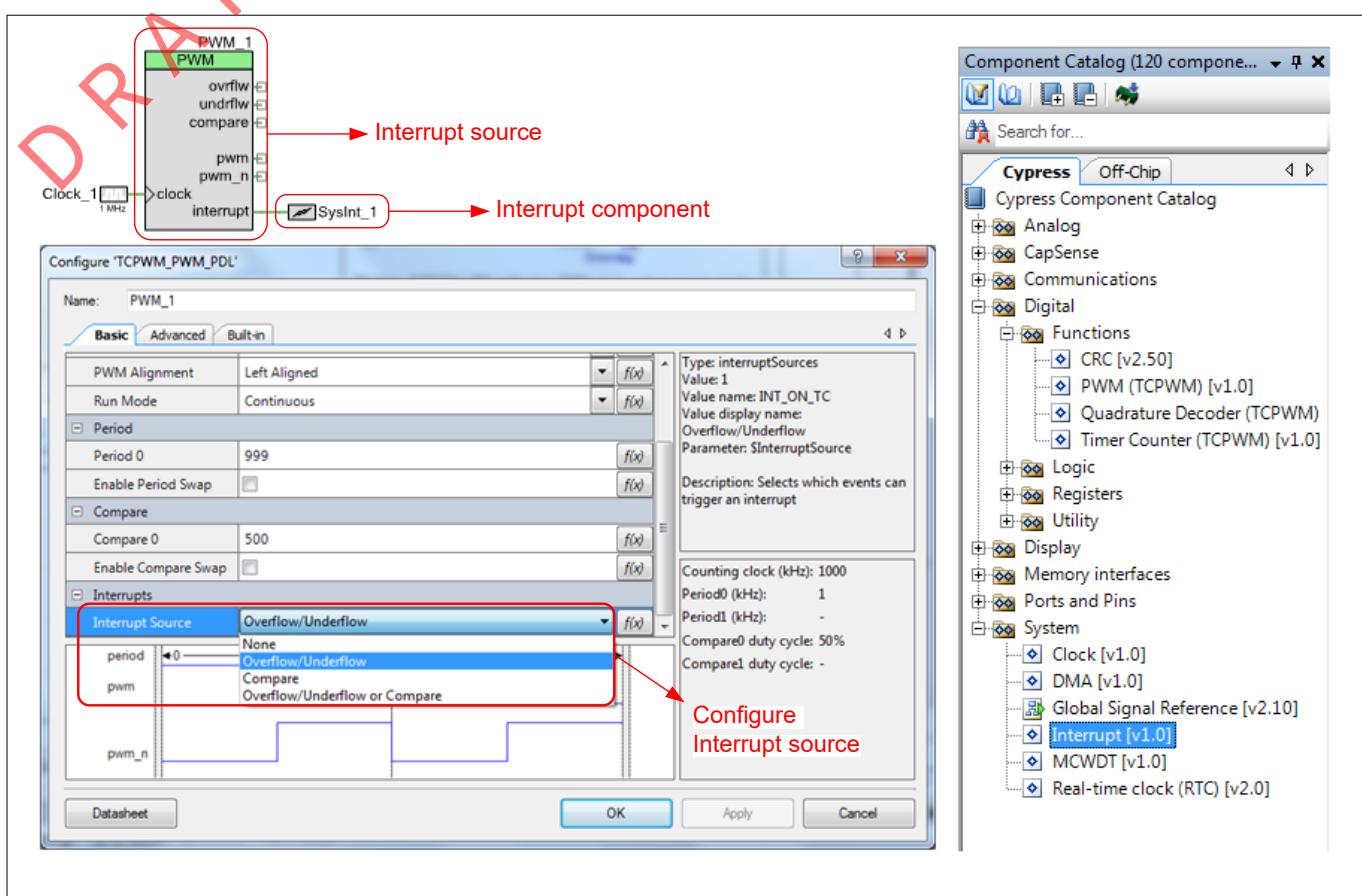


Figure 477 TopDesign with interrupt component SysInt (Interrupt) configuration

Some peripherals do not have an external interrupt terminal (e.g., SCB has interrupts built-in) or may have an option to expose it (e.g., UART).

The Interrupt Component has two configurable options as seen in Figure 478.

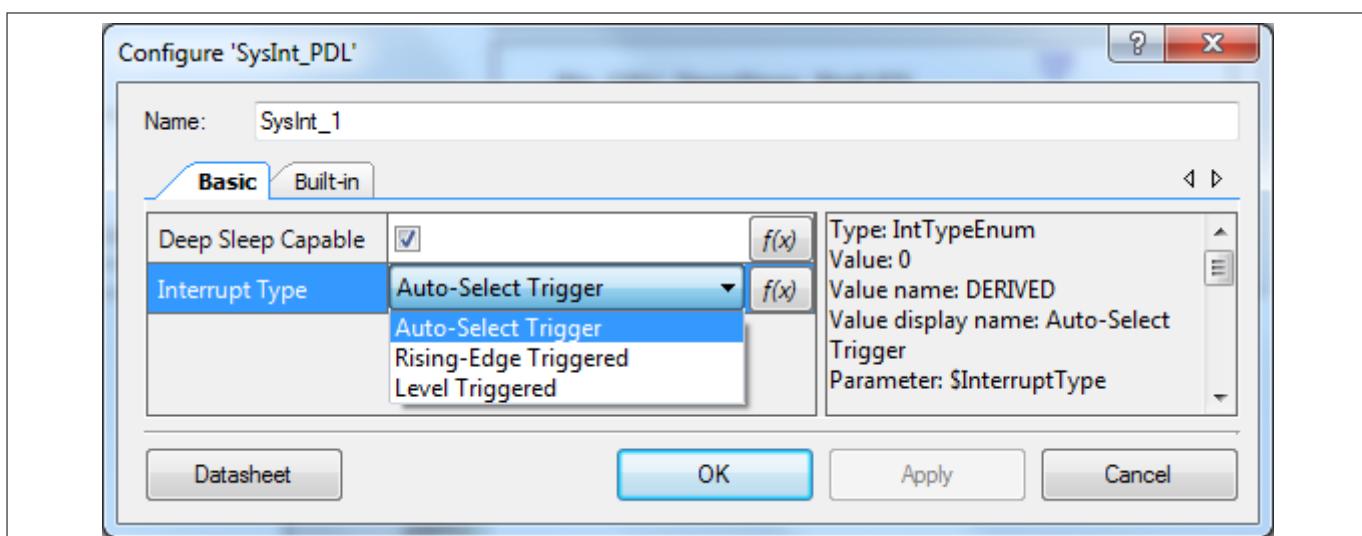


Figure 478 SysInt (Interrupt) configuration

~~5 PSoC™ 6 application notes~~

~~Deep sleep capable~~

Enable this checkbox if you want the interrupt to be assigned to a CPU IRQ line that is deep sleep-capable. You must ensure that the interrupt source is also active and capable of providing the interrupt signal during deep sleep, failing which PSoC™ Creator throws an error when the project is built. Note that this option is significant only in case the interrupt is assigned to CM0+ which has 8 (IRQ 0-7) deep sleep slots to route to. The checkbox is provided only for guidance in automatically assigning an IRQ for the interrupt and can be overridden by manual assignment from the [CyDWR window](#). For CM4, if the interrupt source is deep sleep-capable (IRQ 0-40), disabling the checkbox has no effect on the deep sleep functionality of the interrupt.

Interrupt type

There are three options available for Interrupt type in the Interrupt Component configuration: Auto-Select Trigger, Rising-Edge Triggered, and Level Triggered. The selection of a particular option depends on the interrupt source (fixed-function or UDB/DSI) and the application requirements. In most cases, leave the option to Auto-Select to let PSoC™ Creator derive the interrupt type from the nature of the interrupt source.

Choose only level-triggered for Fixed-function interrupt sources. Choose Level-triggered or Rising-Edge for UDB sources.

5.12.3.3.2 Using the design-wide resource window (CyDWR)

The design-wide resources window (.cydwr file) of the PSoC™ Creator project has an Interrupts tab. This tab lists the instance names of all interrupts used in the TopDesign schematic along with their interrupt numbers.

Each interrupt can be allocated to either CM0+ or CM4 or both the CPUs using the 'ARM CMx Enable' checkbox. Unless specified otherwise, all interrupts are assigned to CM4 by default. **Though possible, it is not advised to assign an interrupt to both CPUs unless an application requires it. A warning icon appears in the Instance name column if both CPUs handle the same interrupt.** A tooltip description of the warning can be viewed on hovering the mouse pointer over the icon.

For CM0+, also assign a CPU IRQ line using the 'ARM CM0+ Vector' column. Note that **some CM0+ IRQs are reserved**. PSoC™ Creator does not allow assigning to these IRQs and will display a warning if done so. There is no option to select the vector for CM4 as these are directly mapped to the corresponding interrupt numbers.

Once assigned to the CPU, assign the priority using the corresponding priority field. CM0+ priority is in the range of 1 to 3, (**priority 0 is reserved** for system calls). CM4 priority is in the range 0 to 7. For both CPUs, priority 0 corresponds to the highest priority and higher numbers denote lower priorities.

A deep sleep-capable interrupt source or IRQ is indicated using an icon . An info icon  appears if a non-deep sleep-capable interrupt is assigned to a deep sleep-capable IRQ line. A build is required to refresh the interrupt numbers and icons.

5 PSoC™ 6 application notes

DRAFT

Instance Name	Interrupt Number	ARM CM0+ Enable	ARM CM0+ Priority (1 - 3)	ARM CM0+ Vector (3 - 29)	ARM CM4 Enable	ARM CM4 Priority (0 - 7)	
EZI2C_1_SCB_IRQ	41	<input checked="" type="checkbox"/>	3	3	<input checked="" type="checkbox"/>	7	
SysInt_1	122	<input checked="" type="checkbox"/>	3	9	<input type="checkbox"/>	--	
SysInt_2	22	<input checked="" type="checkbox"/>	3	4	<input checked="" type="checkbox"/>	--	
UART_1_SCB_IRQ	42	<input type="checkbox"/>	--	--	<input type="checkbox"/>	--	

Legend:

- Warning symbol displayed when interrupt is assigned to both cores
- Warning symbol displayed when a non-Deep Sleep capable interrupt is assigned to a Deep Sleep capable IRQ line
- Warning symbol displayed if an interrupt is not assigned to any core
- Icon indicating Deep Sleep capable interrupt source or vector

Pins Analog DMA Clocks Interrupts System Directives

Figure 479 Interrupts assignment in CyDWR

5.12.3.3.3 Using PSoC™ Creator generated code and PDL

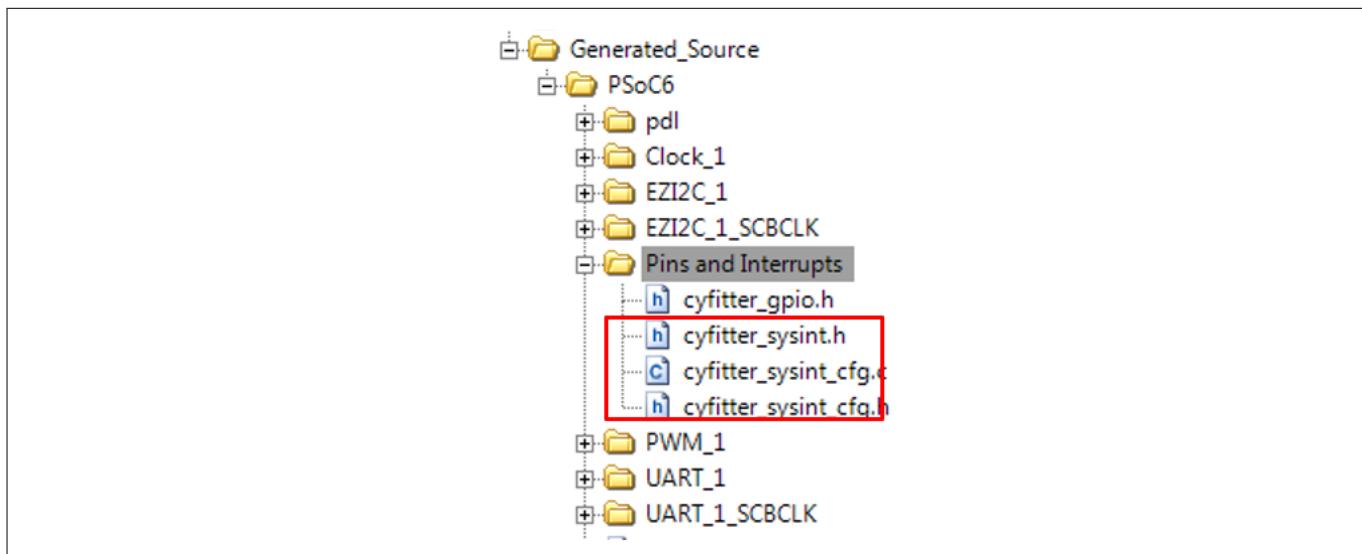


Figure 480 Generated files

Building the project generates code for use in the application. The **Pins and Interrupts** folder contains files with code generated using the information entered in the Interrupts tab in CyDWR.

Cyfitter_sysint.h contains macros with information on interrupt number, its CPU assignment, and priority.

Cyfitter_sysint_cfg.c/h declares and pre-populates instances of SysInt PDL configuration structure using the CyDWR information.

The configuration structure for each interrupt is conditionally defined based on the CPU assignment.

The steps to enable interrupts in firmware are similar to the ones listed in the [PDL section](#) but fewer in number.

- Call the `__enable_irq()` API to enable global interrupts
- Call `Cy_SysInt_Init(&SysInt_1_cfg, ISR_1_handler)`
 - Where `SysInt_1_cfg` is the name of the auto-generated structure from the `cyfitter_sysint_cfg.c` file. `ISR_1_handler` is the name of the interrupt handler that executes when the interrupt triggers. The handler function can reside in the respective CPU's main.c to which the interrupt is assigned. If the

5 PSoC™ 6 application notes

~~DATE~~ handler exists outside main.c, that file must be compiled and linked into the executable for the CPU that handles the ISR

- This step configures the interrupt (routing, priority, and interrupt handler assignment) but does not enable it
- Call NVIC_ClearPendingIRQ(SysInt_1_cfg.intrSrc) to clear any pending interrupts
- Call NVIC_EnableIRQ(SysInt_1_cfg.intrSrc) to enable the interrupt

You can use PSoC™ Creator to generate code, and import that into a preferred IDE. [AN219434 – Importing PSoC™ Creator Code into an IDE for a PSoC™ 6 MCU Project](#) describes how to do that. It is recommended that you use PSoC™ Creator to set up and configure interrupts in PSoC™ 6 MCU, export the project to the IDE you prefer and continue developing firmware code with the IDE preferred.

5 PSoC™ 6 application notes

5.12.4 Debugging tips

This section provides tips on trouble-shooting and debugging interrupts. The following are some of the frequently encountered cases:

• **Interrupt is not triggered**

- Ensure that the interrupt source and global interrupt are enabled
- Ensure that the interrupt vector is initialized with correct ISR
- Check whether other interrupt sources are triggered repeatedly, so consuming the entire CPU bandwidth

• **Interrupt is triggered repeatedly**

This can happen in multiple cases: Insert breakpoints in the ISR and elsewhere in the program which is expected to execute repeatedly (for example, the super-loop in the main function). If the program is not entering the main function, interrupt is triggered repeatedly

- The interrupt line from a fixed-function source

Resolution: Clear the interrupt source to resolve this behavior

- A digital output from the Component (not the interrupt line) is connected to a SysInt Component configured to level type in PSoC™ Creator

Resolution: Configure the Interrupt Component to rising for example to get one interrupt per rising edge.

• **Execution of the ISR is taking longer than expected**

This can happen if other high-priority interrupts are triggered during the execution of the ISR.

Resolution: Increase the priority of the interrupt relative to other interrupt sources.

The [PSoC™ 6 BLE Pioneer Kit](#) has the KitProg2 onboard programmer/debugger.

The [CY8CPROTO-062-4343W PSoC™ 6 Wi-Fi BT Prototyping Kit](#) has the KitProg3 onboard programmer/debugger. This kit is supported only on ModusToolbox™.

PSoC™ Creator supports debugging one CPU at a time (either CM0+ or CM4). ModusToolbox™ IDE supports debugging of both CPUs simultaneously.

The debug mode is useful for checking interrupts as given below:

- To check if an interrupt is executing, add a breakpoint at one of the instructions in the ISR
- Use Breakpoint Hit Count/breakpoint condition to detect the number of times an interrupt is triggered. This is particularly useful to check if the interrupt signal has glitches causing the interrupt to trigger multiple times. To see Breakpoint Hit Count, right-click on the breakpoint, select Hit Count and observe current hit count.
- Use the Call Stack window of the debugger to check program flow to learn when a particular ISR is executed. You can also use it to check if a high-priority interrupt occurred during the execution of a low-priority ISR
- As an alternative to the debugger, you can also use a pin to do the following:
 - Check if the CPU is entering the ISR
 - Measure the ISR execution time. This can be done, for example, by asserting the pin in the beginning of the ISR and de-asserting the pin before returning from the ISR. The time for which the pin is HIGH can be measured using an oscilloscope to give the duration of ISR execution

5 PSoC™ 6 application notes

DRAFT

5.12.5 Advanced interrupt topics

5.12.5.1 Exceptions

Exceptions are the events that cause the processor to suspend the currently executing code and branch to a handler. Interrupts are a subset of exceptions. Besides interrupts, exceptions exist for operating system applications and fault handling.

Exception	Exception number	Exception priority	CPUs supporting the exception	Description
Reset	1	-3	Both CM0+ and CM4	<p>This exception can occur due to multiple reasons, such as power-on-reset (POR), external reset signal on XRES pin, or watchdog reset.</p> <p>Cortex®-M4 execution begins only after CM0+ de-asserts the M4 reset.</p> <p>The reset exception address in the SRAM vector table will never be used because the device comes out of reset with the flash vector table selected. The register configuration to select the SRAM vector table can be done only as part of the startup code in flash after the reset is de-asserted.</p>
Nonmaskable Interrupt (NMI)	2	-2	Both CM0+ and CM4	<p>Both CPUs have their own NMI exception. NMI can be triggered by the following: Any of the interrupt sources, by setting NMIPENDSET bit or using System Calls.</p> <p>PSoC™ 6 BLE supports routing of only one system interrupt as the source for NMI.</p> <p>CY8C62x8/A supports four system interrupt sources for NMI. The four selected interrupt sources are logically Ored into a single CPU NMI input</p> <p>NMI exception handler address is automatically initialized to the system call API located in SROM (at 0x0000000D by the boot code. The value should be retained by the user during vector table relocations; otherwise, no system call will be executed.</p>
HardFault Exception	3	-1	Both CM0+ and CM4	HardFault exception occurs when executing an undefined instruction or accessing an invalid memory addresses.
SVCall Exception	11	Configurable	Both CM0+ and CM4	Supervisor Call (SVCall) is an always-enabled exception caused when the CPU executes the SVC instruction as part of the application code. The SVC instruction enables the application to issue a supervisor call that requires privileged access to the system.

5 PSoC™ 6 application notes

Exception	Exception number	Exception priority	CPUs supporting the exception	Description
PendSV	14	Configurable	Both CM0+ and CM4	PendSV exception is normally software-generated. PendSV is another supervisor call related exception similar to SVCal.
SysTick Exception	15	Configurable	Both CM0+ and CM4	SysTick is a 24-bit decrementing counter that generates periodic interrupts.
Memory Management Fault Exception	4	Configurable	Only CM4	A memory management fault is an exception that occurs because of a memory protection-related fault.
Bus Fault Exception	5	Configurable	Only CM4	A Bus Fault is an exception that occurs because of a memory-related fault for an instruction or data memory transaction.
Usage Fault Exception	6	Configurable	Only CM4	A Usage Fault is an exception that occurs because of a fault related to instruction execution.

Notes:

3. Exception priority that are configurable can be configured from priority 0-3 for CM0+ and 0-7 for CM4.
4. Interrupts are also part of exceptions. Interrupt vector number 0 (i.e., IRQ 0) corresponds to the exception number 16, and so on.

5.12.5.2 Interrupt latency

Interrupt latency is defined as the time delay between the assertion of an interrupt and the execution of the first instruction in its ISR. CM0+ has a latency of 15 clock cycles (worst case); CM4 has a latency of 12 clock cycles (worst case). Some peripherals generate additional cycles due to synchronization circuit between the peripherals and CPUs. [Table 97](#) provides the number of CPU clock cycle delays for various peripherals in PSoC™ 6 MCU.

Table 97 Synchronization delay for various peripherals

Interrupt source	Synchronization delay
TCPWM, DMA, USB, I2S, PDM – PCM, CDS	0 clock cycles
SCB, GPIO, LPComp, RTC, WDT, SMIF, BLE	2 clock cycles

When both CPUs are in sleep/deep sleep power mode, there is a need for additional two clock cycles required for synchronization.

Context switching affects the latency and involves the following steps:

- Current instruction execution is completed
- The processor pushes the current Program Counter (PC), Link Register (LR), Program Status Register (PSR), and some of the general-purpose registers (Program and Status Register (PSR), Return Address, Link Register (LR or R14), R12, R3, R2, R1, and R0) to the stack
- The processor reads the vector address from the NVIC and updates it to the PC
- The processor updates the NVIC registers

~~5 PSoC™ 6 application notes~~

Thus, the latency varies depending on the current instruction being executed. To make the process efficient, both CM0+ and CM4 processors implement the following two schemes:

Tail Chaining: If an interrupt is in the pending state while the processor is executing another interrupt handler, unstacking is skipped when the execution ends for the first interrupt and the handler for the pending interrupt is immediately executed. This saves the time of restoring the registers from the stack and pushing the same registers again to stack. This is useful for nested interrupts, as seen in the following section, and for reducing the latency of low-priority interrupts.

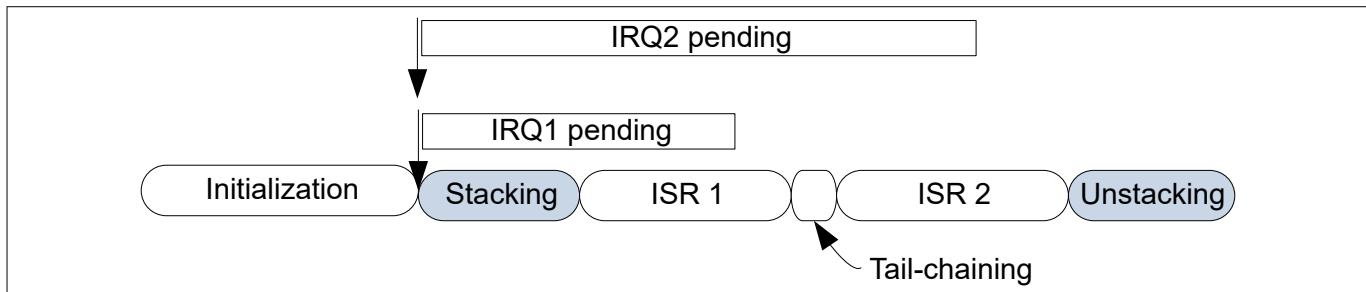


Figure 481 Tail chaining

Late Arrival: If a higher-priority interrupt occurs during the stacking process of a lower-priority interrupt, the processor jumps to the higher-priority interrupt handler instead of a lower-priority one. The processor reads the vector address of the higher-priority interrupt at the end of the stacking process. Once the higher-priority interrupt handler execution is completed, the vector address for the pending lower-priority interrupt handler is fetched and executed. This reduces the latency for a higher-priority interrupt by entering the lower priority ISR and pushing the register values to the stack.

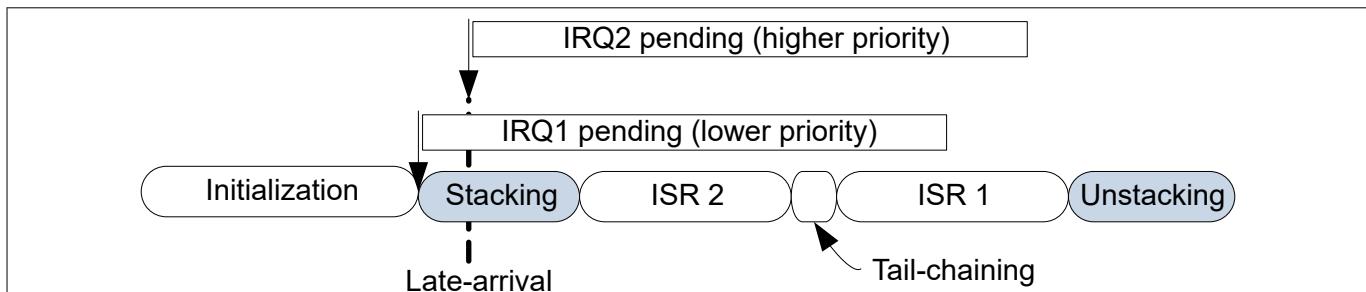


Figure 482 Late arrival

5.12.5.3 Nested interrupts

NVIC automatically handles nested interrupts without any software overhead. If a higher-priority interrupt is asserted during the execution of a lower-priority interrupt handler, some of the general-purpose registers are pushed to stack, CPU reads the vector address from NVIC and jumps to the higher-priority interrupt handler. After the execution is completed, the processor restores the register values and execution resumes for the lower-priority interrupt.

5.12.5.4 Code optimization

An important performance requirement in interrupt-based applications is the ISR code execution time. In some applications, the critical code in the ISR must be executed within a particular time of receiving the interrupt request. Also, interrupt execution should not take too much time and stall the main code execution or other interrupts. To meet these requirements, use the following guidelines:

- Avoid calls to lengthy functions in the ISR. Functions such as Character LCD display routines or printing long strings to a UART terminal takes long time to execute, so blocking the execution of other low-priority interrupts. The recommended technique is to move non-critical function calls to the main code and just

5 PSoC™ 6 application notes

- ~~DRAFT~~
- set a flag variable in the ISR. The main code periodically checks the flag and if set, clears it and calls the function
 - Assign proper priority to the interrupts. In applications with multiple interrupts, give a higher priority to more time-critical interrupts

Although [AN89610 – PSoC™ 4 and PSoC™ 5LP ARM Cortex® Code Optimization](#) targets a different CPU architecture, it is a useful reference for general compiler topics.

5 PSoC™ 6 application notes

References

-
- 1
- 2

For a complete and updated list of PSoC™ 6 MCU code examples, visit our [code examples web page](#). For more PSoC™ 6 MCU-related documents, please visit our [PSoC™ 6 MCU product web page](#).

Table 98 Documents related to PSoC™ 6 MCU features

Document	Document name
Application notes	
AN228571	Getting Started with PSoC™ 6 MCU on ModusToolbox™
AN221774	Getting Started with PSoC™ 6 MCU on PSoC™ Creator
AN210781	Getting Started with PSoC™ 6 MCU with Bluetooth® Low Energy (BLE) Connectivity
AN215656	PSoC™ 6 MCU Dual-CPU System Design
AN219434	Importing PSoC™ Creator Code into an IDE for a PSoC™ 6 MCU Project

Programmable digital

Code Examples (ModusToolbox™) on GitHub

mtb-example-psoc6-gpio-interrupt	PSoC™ 6 MCU GPIO Interrupt
mtb-example-psoc6-hello-world	PSoC™ 6 MCU Hello World Example
mtb-example-psoc6-uart-transmit-receive-dma	PSoC™ 6 MCU: SCB UART Transmit and Receive with DMA
mtb-example-psoc6-spi-master-dma	PSoC™ 6 MCU: SCB SPI Master with DMA
mtb-example-psoc6-wdt	PSoC™ 6 MCU Watchdog Timer
mtb-example-psoc6-capsense-buttons-slider	PSoC™ 6 MCU: CAPSENSE™ Buttons and Slider
mtb-example-psoc6-csdadc	PSoC™ 6 MCU: CSD Analog-to-Digital Converter (ADC)
mtb-example-psoc6-capsense-custom-scan	PSoC™ 6 MCU: CAPSENSE™ Custom Scan
mtb-example-psoc6-capsense-buttons-slider-freertos	PSoC™ 6 MCU: CAPSENSE™ Buttons and Slider (FreeRTOS)
mtb-example-psoc6-ble-findme	PSoC™ 6 MCU with Bluetooth™ Low Energy (BLE) Connectivity: Find Me Application
mtb-example-psoc6-ble-battery-level-freertos	PSoC™ 6 MCU with BLE Connectivity: Battery Level (FreeRTOS)
mtb-example-psoc6-ble-throughput-freertos	PSoC™ 6 MCU: BLE Throughput Measurement (FreeRTOS)
mtb-example-psoc6-usb-audio-device-freertos	PSoC™ 6 MCU: USB Audio Device with FreeRTOS
mtb-example-psoc6-usb-cdc-echo	PSoC™ 6 MCU: USB CDC Echo Application
mtb-example-psoc6-usb-hid-mouse	PSoC™ 6 MCU: USB HID Mouse Application
mtb-example-psoc6-usb-hid-generic	PSoC™ 6 MCU: USB HID Generic Application
mtb-example-psoc6-usb-msc-file-system	PSoC™ 6 MCU: USB Mass Storage File System
mtb-example-psoc6-usb-msc-logger	PSoC™ 6 MCU: USB Mass Storage Logger
Code Examples (PSoC™ Creator)	
CE219521	PSoC™ 6 MCU - GPIO Interrupt

(table continues...)

Reference manual

5 PSoC™ 6 application notes

Table 98 (continued) Documents related to PSoC™ 6 MCU features

Document	Document name
CE216795	PSoC™ 6 MCU Dual-Core Basics
CE218129	PSoC™ 6 MCU Wake up from hibernate Using Low-Power Comparator
CE218542	PSoC™ 6 Custom Tick Timer Using RTC Alarm Interrupt
CE219339	PSoC™ 6 MCU MCWDT and RTC Interrupts (Dual Core)
CE220061	PSoC™ 6 MCU Multi-Counter Watchdog Interrupts
CE220607	PSoC™ 6 MCU Watchdog Timer in interrupt mode
CE220169	PSoC™ 6 MCU Periodic Interrupt Using TCPWM
CE212736	PSoC™ 6 MCU with Bluetooth™ Low Energy (BLE) Connectivity - Find Me

Software/IDE

PSoC™ Creator	PSoC™ Creator User Guide
ModusToolbox™	ModusToolbox™ User Guide
Peripheral Driver Library (PSoC™ Creator)	PDL API Reference available on installation
Peripheral Driver Library (ModusToolbox™)	PDL API Reference

~~5 PSoC™ 6 application notes~~

A Appendix A. Interrupt sources in PSoC™ 6 MCU

For information on IRQ number and applicable power mode for each interrupt, see the “Interrupts” chapter of the respective devices’ technical reference manual.

Interrupt source	Details
GPIOs	<p>Each port consists of a maximum of eight pins. Each pin can generate an interrupt, but the vector address is common for all pins in a port. Firmware must identify the pin that caused the interrupt. PSoC™ 6 MCU enables interrupt trigger on the rising edge, falling edge, or both edges of the GPIO signal. This interrupt can wake the device from sleep, deep-sleep modes.</p> <p>There is a GPIO All Ports interrupt that allows combining all port interrupts into a single vector. Firmware must identify the port that caused the interrupt.</p> <p>There is a GPIO Supply Detect Interrupt that can be used to detect the supply ramping up or ramping down.</p>
LPCOMP	Like GPIOs, an interrupt can be triggered on the rising edge, falling edge, or both edges of the comparator output signal. LPCOMP can also wake the device from sleep, deep sleep, and hibernate power modes.
SCB (deep sleep)	SCB interrupt that can wakeup CPU/system from deep sleep
Multi Counter Watchdog Timer (MCWDT) Interrupt	MCWDT configures two 16-bit counters and one 32-bit counter capable of generating periodic interrupts. MCWDT can wake the CPU from deep sleep power mode.
Backup Domain Interrupt	Backup domain interrupt includes the RTC ALARM1, RTC ALARM2, and RTC century overflow interrupt. This can be used to wake the CPU from sleep, deep sleep, and hibernate power modes.
Other Combined Interrupts for SRSS	<p>The following cases generate this interrupt: WDT interrupt, Low Voltage Detect (LVD) interrupt, and clock calibration interrupt. WDT interrupt occurs when the watchdog counter value matches the preset Counter Match value. Missing two interrupts will cause a watchdog reset.</p> <p>Low-voltage detect (LVD) interrupt when the device supply voltage drops below a threshold.</p> <p>Clock calibration interrupt is triggered when clock calibration is complete.</p> <p>These are capable of waking the CPU from deep sleep.</p>
CTBm Interrupt (all CTBms)	This block provides continuous time analog functionality. It generates interrupts on event such as comparator triggers.
Bluetooth Radio Interrupt	Bluetooth sub-system interrupt
IPC Interrupt	IPC interrupts could be triggered when an IPC release or notify event occurs.

~~5 PSoC™ 6 application notes~~

Interrupt source	Details
SAR ADC	
CTBm Interrupt (individual CTBm)	This block provides continuous time analog functionality. It generates interrupts on event such as comparator triggers.
PASS FIFO Interrupt	
SCB	<p>PSoC™ 6 MCU supports SCBs which can be configured as SPI, I²C or UART. One SCB interrupt amongst the 8 SCBs is deep sleep-capable. The following events generate an interrupt in a SCB.</p> <ul style="list-style-type: none"> TX FIFO has less entries than specified. TX FIFO is not full/full/overflow/underflow. RX FIFO has more entries than the value specified, RX FIFO is full/not empty. SPI: SPI interrupts are triggered when SPI master transfer done, SPI Bus Error, SPI slave deselected after any EZSPI transfer occurred I²C: I²C master lost arbitration, received NACK, received ACK, sent STOP, I²C bus error, I²C slave lost arbitration, received NACK, received ACK, received STOP, received START, address matched. UART Interrupts: TX received a NACK in SmartCard mode, TX done, Arbitration lost, frame error in received data frame, parity error in received data frame, LIN baud rate detection is completed, LIN break detection is successful
CSD (CAPSENSE™) Interrupt	CSD, used for touch applications, generates an interrupt when the sensor scan is complete.
CPUSS DMAC, Channel #0 – 3	DMA Controller (DMAC) interrupts on DMA events like transfer completion, bus errors, address misalignments, current pointer being NULL, active channel disabled and descriptor error.
DMA Interrupt	DMA interrupt can be generated when the data transfer is completed.
CPUSS Fault Structure Interrupt #0	This interrupt occurs when there is a protection unit access violation.
CRYPTO Accelerator Interrupt	Crypto Interrupt is generated in the following cases: When a FIFO event is activated, FIFO overflows, true random number generator is initialized, true random number generator has generated a data value of the specified bit size, pseudo random number generator has generated a data value, instruction decoder encounters an instruction with a non-defined operation code, instruction decoder encounters an instruction with a non-defined condition code, when a AHB-Lite bus error is observed, true random number generator monitor adaptive proportion test detects a repetition of a specific bit value, true random number generator monitor adaptive proportion test detects a disproportionate occurrence of a specific bit value.
FLASH Macro Interrupt	Flash controller has a timer that generates interrupts.

~~5 PSoC™ 6 application notes~~

Interrupt source	Details
Floating point interrupt	Floating Point operation fault
CM0+ CTI #0	CTI triggers are used to communicate events between debug components.
CM4 CTI #0	CTI triggers are used to communicate events between debug components.
TCPWM	The TCPWM block can be configured to work as a 16- or 32-bit timer, counter, or PWM. It can generate interrupts on terminal count, input capture signal, or a compare true event.
UDB Interrupt #0	Any digital signal generated in a UDB can trigger an interrupt. Signals are routed to the interrupt controller through the routing fabric known as Digital System Interconnect (DSI).
I2S Audio Interrupt	Interrupt can be generated in the following cases. Less entries in the TX FIFO than the value specified, TX FIFO is not full, TX FIFO is empty, attempt to write to a full TX FIFO, attempt to read from an empty TX FIFO, triggers when the Tx watchdog event occurs, more entries in the RX FIFO than the value specified , RX FIFO is not empty, RX FIFO is full, attempt to write to a full RX FIFO, attempt to read from an empty RX FIFO, triggers when the Rx watchdog event occurs.
PDM/PCM Audio interrupt	More entries in the RX FIFO than the value specified, RX FIFO is not empty, attempt to write to a full RX FIFO, attempt to read from an empty RX FIFO
Energy Profiler Interrupt	This interrupt occurs on a profiling counter overflow.
Serial Memory Interface Interrupt	This interrupt is activated when TX data FIFO is activated, RX data FIFO is activated, alignment error, FIFO overflow.
USB Interrupt	The USB block has a predefined set of 13 interrupt trigger events that can be mapped to either one of the three interrupts. Events such as USB Start of Frame (SOF), USB bus reset, data endpoint events, control endpoint events, Arbiter Interrupt Event, and Link Power Management (LPM) event generate interrupts.
CAN Interrupt	CAN consolidated or individual channel interrupt
Consolidated Interrupt for all DACs	Interrupt can be generated when DAC buffer is empty. This interrupt can be used by the CPU to transfer the next value to the DAC.
SDIO wakeup Interrupt for SDHC	SDIO wakeup interrupt triggered on events such as card insertion, removal, and SDIO card interrupt. This doesn't wakeup the system from deep sleep.
Consolidated Interrupt for SDHC	Consolidated interrupt on all other normal/error events related to SDHC
EEMC Wakeup Interrupt for mxsdhc, not used	EEMC wakeup interrupt for SDHC block (reserved)
Consolidated Interrupt for SDHC	Consolidated interrupt (reserved)

5 PSoC™ 6 application notes
DRAFT

5.12.7 Revision history

Document version	Date of release	Description of changes
**	2017-09-15	New Application Note
*A	2018-11-08	Updated for CY8C62x8/A and ModusToolbox™
*B	2019-02-25	Updated section “Configuring Interrupts Using ModusToolbox™”
*C	2019-10-21	Sunset review Updated section “Configuring Interrupts Using ModusToolbox™”
*D	2020-07-29	Updated information for CY8C61x6/7, CY8C62x4, and CY8C62x5 devices Updated References
*E	2021-03-27	Migrated to new template.
*F	2022-07-21	Template update

5.13 AN218241 PSoC™ 6 MCU hardware design considerations

About this document

-
- 1
- 3

Scope and purpose

AN218241 shows you how to design a hardware system around a PSoC™ 6 MCU device, starting with considerations for package selection, power, clocking, reset, I/O usage, programming and debugging interfaces, and analog module design.

Associated part family

[PSoC™ 6 MCU](#)

5 PSoC™ 6 application notes~~DRAFT~~
5.13.1 Introduction

PSoC™ 6 MCU is Infineon ultra-low-power PSoC™ device with a dual-CPU architecture tailored for smart homes, IoT gateways, and so on. The PSoC™ 6 MCU device is a programmable embedded system-on-chip that integrates the following features on a single chip:

- Single-CPU microcontroller: Arm® Cortex®-M4 (CM4) or Dual-CPU microcontroller: Arm® Cortex®-M4 (CM4) and Cortex®-M0+ (CM0+)
- Programmable analog and digital peripherals
- Up to 2 MB of flash and 1 MB of SRAM
- Fourth-generation CAPSENSE™ technology

This application note discusses considerations for hardware design including package, power, clocking, reset, I/O usage, programming/debugging, CAPSENSE™, and BLE antenna design.

The PSoC™ 6 MCU device must be configured to work in its hardware environment, which you can do with integrated design environment (IDE) like PSoC™ Creator™ or ModusToolbox™ IDE. The application note explains various configurations available in PSoC™ Creator and ModusToolbox™ IDE required to set up the device for a given hardware environment.

To get started with PSoC™ 6 MCU, see [AN228571 – Getting Started with PSoC™ 6 MCU on ModusToolbox™](#) or [AN221774 – Getting Started with PSoC™ 6 MCU on PSoC™ Creator](#).

5 PSoC™ 6 application notes
~~DRAFT~~

5.13.2 Package selection

One of the first decisions that you must make for your PCB is the choice of package. Several considerations drive this decision, including the number of PSoC™ device pins required, PCB and product size, PCB design rules, and thermal and mechanical stability. PSoC™ 6 MCU devices are available in BGA, WLCSP, MCSP, QFN, and TQFP packages – see [Table 99](#). See the respective device [datasheet](#) for more details on packaging.

Table 99 **Package dimensions**

PSoC™ 6 MCU device	Package
CY8C61x6/7	124-BGA
	80-WLCSP
CY8C62x6/7	124-BGA
	80-WLCSP
CY8C63x6/7	116-BGA
	104-MCSP
	124-BGA
	68-QFN
CY8C61xA/8	124-BGA
CY8C62xA/8	128-TQFP
	100-WLCSP
CY8C61x5	100-TQFP
CY8C62x5	68-QFN
	49-WLCSP
CY8C62x4	80-TQFP
	64-TQFP
	68-QFN

As a design reference, see 6 MCU CAD Libraries [PSoC™ 6 MCU CAD Libraries](#), which contain PSoC™ 6 MCU schematics and PCB libraries. Note that you may need to modify the libraries slightly when you use them in your hardware design. Infineon takes no responsibility for issues related to the use of the libraries.

~~5 PSoC™ 6 application notes~~

~~5.13.3 Power~~

PSoC™ 6 MCU can be powered by a single supply with a wide voltage range, from 1.7 V to 3.6 V. As listed in [Table 100](#), it has separate power domains for analog and digital modules. Details of connections to these pins are discussed in the [Power pin connections](#) section.

5.13.3.1 Buck regulator inductor/capacitor selection

PSoC™ 6 MCU has an on-chip buck regulator. In some PSoC™ 6 MCU devices, the buck regulator generates two outputs (VBUCK1 and VRF). The VBUCK1 output can power the PSoC™ 6 MCU core. The VRF output can power the BLE radio in CY8C63x6/7 devices. In other PSoC™ 6 MCU devices, the on-chip buck regulator generates one output that can power the MCU core. See [Table 100](#) to know the type of buck regulator supported by the PSoC™ 6 MCU device selected for your design.

When selecting the inductor and capacitor for the buck regulator, it is recommended to use the following components:

Inductor: Use an inductor with an inductance value of 2.2 μ H. It is recommended to keep the DC resistance below 0.2 Ω .

Capacitor: The capacitor connected to the output of the buck regulator should meet the required capacitance at the operating voltage (1.1 V or 0.9 V) that is, 4.7 μ F load capacitor for VCCD pin. For BLE operation in CY8C63x6/7 devices, a load capacitor of 10 μ F at VRF pin or VDCDC pin is required. See [Figure 483](#) and [Figure 484](#). It is recommended to use a load capacitor with dielectric constant of X5R or above.

See the "Power Supply and Monitoring" chapter of [PSoC™ 6 MCU: Architecture TRM](#) for details on the buck regulator and how to use it.

Table 100 PSoC™ 6 MCU power domains

Power domain	Supply pin ³⁵⁾	Ground pin	Supported voltage range	Description	Supported Devices
SIMO Buck Regulator	VDD_NS ³⁶⁾	VSS	1.7 V to 3.6 V	Input to the SIMO Buck Regulator	CY8C61x6/7, CY8C62x6/7, CY8C63x6/7 devices
	VBUCK1 ³⁶⁾	VSS	ULP or LP	Output 1 of on-chip SIMO regulator. Requires bypass capacitor connection for proper operation. This output can power VCCD when internal regulators are powered down. See the device datasheet for supported voltage range (ULP or LP).	
	VRF ³⁶⁾	VSS	1.05 V to 1.50 V	Output 2 of on-chip SIMO regulator. Requires bypass capacitor connection for proper operation. This output can power the VDCDC input.	
	VIND1 ³⁶⁾ , VIND2 ³⁶⁾	–	–	Pins for inductor required by the SIMO buck regulator	
SISO Buck Regulator	VDD_NS ³⁶⁾	VSS	1.7 V to 3.6 V	Input to the SISO buck regulator	CY8C61xA/8, CY8C62xA/8, CY8C61x5,

(table continues...)

³⁵ Availability of supply pins in various power domains depends on the device. See the respective device datasheets for more details.

³⁶ Optional supply pins. When these supply rails (input or output) are not used, it is recommended to leave the pin floating. For example, when USB is not used, you can leave VDDUSB supply pin floating.

5 PSoC™ 6 application notes

DRAFT Table 100 (continued) PSoC™ 6 MCU power domains

Power domain	Supply pin ³⁵⁾	Ground pin	Supported voltage range	Description	Supported Devices
	VIND ³⁶⁾	–	–	Inductor connection for the internal buck regulator	CY8C62x5, CY8C62x4 devices
Analog	VDDA	VSS	1.7 V to 3.6 V	Analog power supply input	All PSoC™ 6 MCU devices
Digital	VDDD	VSS	1.7 V to 3.6 V	Digital power supply input and core regulators' supply input	All PSoC™ 6 MCU devices
	VCCD	VSS	ULP or LP	Internal core regulators' (LDO) output. Requires bypass capacitor connection for proper operation. Used as core supply input when internal regulators are OFF. Refer to device datasheet for supported voltage range (ULP or LP).	
	VDDUSB ³⁶⁾	VSS	2.85 V to 3.6 V	Power supply pin for USB block. When USB is not used, the pin can support voltage ranging from 1.7 V to 3.6 V	PSoC™ 6 MCU devices with USB support
I/O	VDDIO _x where x = 0,1,2	VSS	1.7 V to 3.6 V	I/O power supply input	All PSoC™ 6 MCU devices
RF	VDDR ³⁶⁾	VSSR	1.05 V to 1.50 V	BLE radio analog supply input; connected to VDCDC externally	CY8C63x6, CY8C63x7 devices
	VDDR_HV ^L	VSS	1.75 V to 1.95 V	PSoC™ 6 MCU to BLE radio interface supply output; requires bypass capacitor connection for proper operation	
	VDCDC	VSS	1.05 V to 1.50 V	BLE radio digital supply input; typically connected to VRF externally	
	DVDD ³⁶⁾	VSS	~ 1 V	BLE subsystem regulators' (LDO) output; requires bypass capacitor connection for proper operation	
Backup	VBACKUP	VSS	1.4 V to 3.6 V	Backup domain supply	All PSoC™ 6 MCU devices

5.13.3.2 Power pin connections

PSoC™ 6 MCU offers power supply options that support a wide range of application voltages and requirements. The VDDD input supports a voltage range of 1.7 V to 3.6 V. If the application voltage is in this range, then PSoC™ 6 MCU can be connected directly to the application voltage. In applications that have the voltage beyond this range, a suitable power management IC (PMIC) should be used to bring the voltage to this range.

³⁵ Availability of supply pins in various power domains depends on the device. See the respective device datasheets for more details.

³⁶ Optional supply pins. When these supply rails (input or output) are not used, it is recommended to leave the pin floating. For example, when USB is not used, you can leave VDDUSB supply pin floating.

~~5 PSoC™ 6 application notes~~

Figure 483 and **Figure 484** show various power pads in PSoC™ 6 MCU and the recommended connections for a typical design. The VBACKUP power pin can operate from 1.4 V to 3.6 V and can exist independent of other rails. Therefore, the VBACKUP rail can be connected to a separate rail such as a coin cell battery or a super capacitor or tied to VDDD directly if the design does not support a separate VBACKUP supply. Other supply rails and pins such as VDDA and VDDIO also exist independent of VDDD and VCCD. In addition, the power rails provide supply voltage to I/O ports. **Table 101** shows the I/O ports and their respective power supply rails. I/O levels on a port should not exceed its supply voltage; otherwise it will be clamped to the supply level.

Table 101 Power supply for I/O ports

I/O ports powered	Power supply rail	Alternate supply rail
Port 0	VBACKUP	VDDD
Port 1	VDDD	-
Port 2, 3, 4	VDDIO2	-
Port 5, 6, 7, 8	VDDIO1	-
Port 9, 10	VDDIOA	VDDA
Port 11, 12, 13	VDDIO0	-
Port 14	VDDUSB	-

Note: Availability of ports depends on the device. See the respective device [datasheets](#) for details on available port pins.

As a rule of thumb, connect one 0.1- μ F and one 1- μ F (10 μ F in some cases; see **Figure 483** and **Figure 484**) ceramic decoupling capacitor to each power supply pin. Note that certain packages have more than one VDDD, VDDA, VDDR, and VDDIO x ($x = 0, 1, 2$) pins. In such cases, where the same power supply pin is brought out to multiple pins, the pins can be shorted externally and can share the decoupling capacitors. For example, in 100 TQFP package of CY8C62x5 device, pin VDDIO0 is brought out to pin 91 and pin 92 of the device. These pins (pin 91 and pin 92) can be shorted together and share the same decoupling capacitors. For package specific power supply consideration, refer to the respective device [datasheets](#). A ferrite bead is recommended between the input supply and the VDD_NS pin for isolating the VDD_NS domain from the supply. This is because of the noise injected on the VDD_NS rail from the SIMO buck operation. See **Figure 483** and **Figure 484** for recommended values of various components used. The PCB trace between the pin and the capacitors should be as short as possible. For more information, see [Appendix A](#).

Note: It is a good practice to check a capacitor's datasheet before you use it, specifically for working voltage and DC bias specifications. With some capacitors, the actual capacitance can decrease considerably when the DC bias is a significant lower percentage of the rated working voltage. Ensure that the capacitance variation is lower for the operating voltage range it serves.

5 PSoC™ 6 application notes

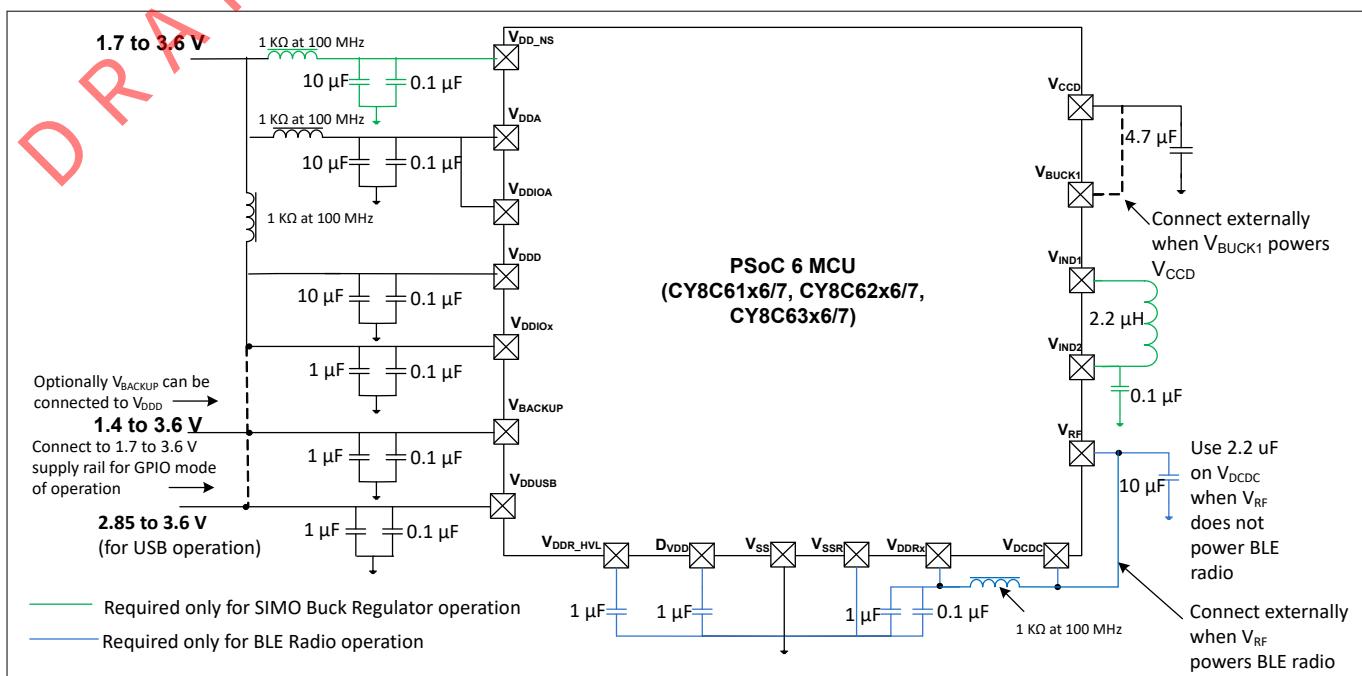


Figure 483 Power system connections

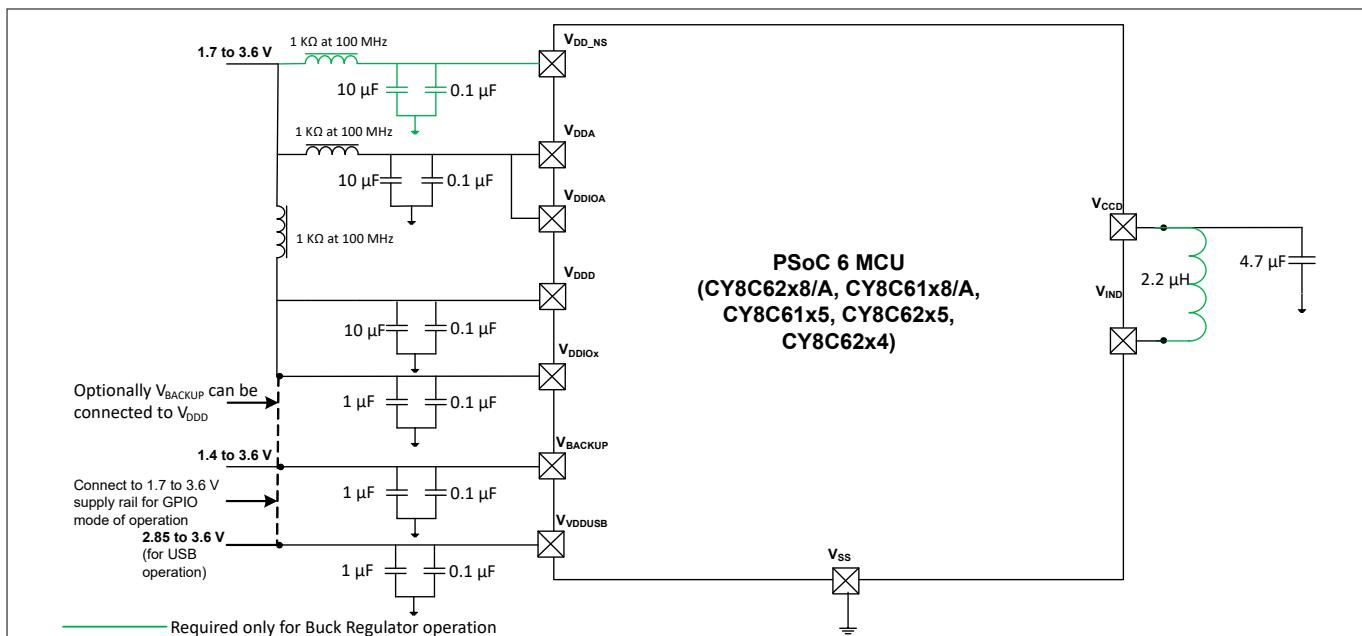


Figure 484 Power system connections

You can use a single power supply rail for digital power and analog power, which helps to simplify the power design in your board. However, to get a better analog performance in a mixed-signal circuit design, use separate power supply rails for the digital power and the analog power. It is recommended to isolate the rails using ferrite beads. For more mixed-signal circuit design techniques, see [AN57821 – PSoC™ Mixed-Signal Circuit Board Layout Considerations](#).

Proper use and layout of capacitors and ferrite beads help to improve the EMC performance. For more information, see [AN80994 – Design Considerations for Electrical Fast Transient \(EFT\) Immunity](#).

The Infineon PSoC™ 6 MCU kits (like [CY8CKIT-062-BLE](#), [CY8CKIT-062-WIFI-BT](#), [CY8CPROTO-062-4343W](#), [CY8CPROTO-062S3-4343W](#) and so on) provide schematics and bills of material (BOMs) that give good examples

5 PSoC™ 6 application notes

~~DRAFT~~

of how to incorporate PSoC™ 6 MCU into board schematics. Also, see the respective device [datasheets](#) for package specific power supply considerations. For more information, see [References](#).

5.13.3.3 PMIC controller

PSoC™ 6 MCU supports a PMIC controller in the backup domain. The PMIC controller can be used to enable/disable the power supplies to PSoC™ with the backup domain (VBACKUP) running from another supply, typically from a coin cell or super capacitor. PSoC™ 6 MCU provides configurable wakeup sources either by an RTC alarm or by a GPIO input pin. To use the PMIC controller, ensure that the following are present in hardware:

The selected PMIC supports an active HIGH PMIC enable signal that supports the input levels in the VBACKUP voltage range.

An independent supply (other than the PMIC output) such as a coin cell or super capacitor powers VBACKUP.

The Wakeup_OUT (P0[5]) pin is connected to the PMIC enable signal with an optional pull-up resistor to VBACKUP.

If the RTC alarm is used as a wakeup source, ensure that an external 32.768-kHz crystal or signal clocks the RTC.

If an external pin (P0[4]) is used as a wakeup source, the signal should be driven externally i.e., a necessary pull-down resistor should be connected externally because the wakeup logic is active HIGH. The Wakeup_IN pin is a HighZ digital input to PSoC™.

Figure 485 illustrates using a PMIC controller with a PSoC™ 6 MCU design.

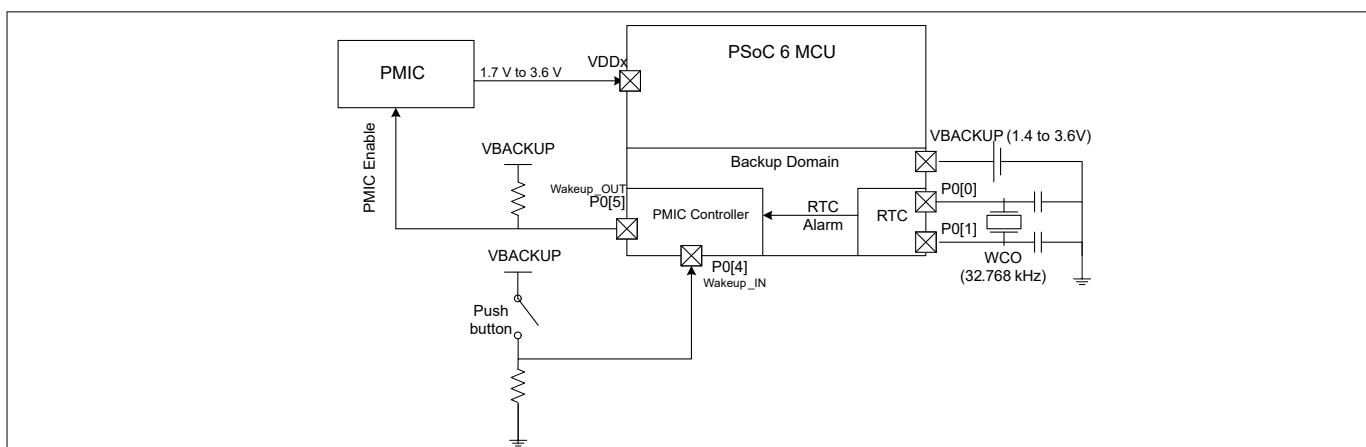


Figure 485 **Using PMIC controller**

5.13.3.4 Power ramp-up and sequencing considerations

VDDD, VBACKUP, VDDIO, and VDDA do not have any sequencing limitation and can establish in any order. However, VDDIO should be greater than or equal to VDDA when using CAPSENSE™ in the design. CAPSENSE™ signals can switch between VSSA and VDDA, so requiring VDDIO to support the swing to VDDA. In addition, the presence of VDDA without VDD or VDDD can cause some leakage from VDDA. However, it will not drive any analog or digital output. All the VDDA pins in packages that offer multiple VDDA supply pins must be shorted externally on the PCB. The maximum allowed voltage ramp rate for any power pin is 100 mV/µs in Active power mode and the allowed ramp is 20 mV/µs in Deep Sleep power mode.

5.13.3.5 Device power settings

You can use Infineon development environments, either PSoC™ Creator or ModusToolbox™ IDE, to manage device power settings. PSoC™ Creator automatically configures Components for optimal performance for the voltages applied to the power pins. To do so, it needs to know the value of these voltages. The System tab in

5 PSoC™ 6 application notes

the PSoC™ Creator project's Design-Wide Resources (DWR) window is used for this purpose. To open the DWR window, double-click the .cydwr file in the project navigator, as Figure 486 shows.

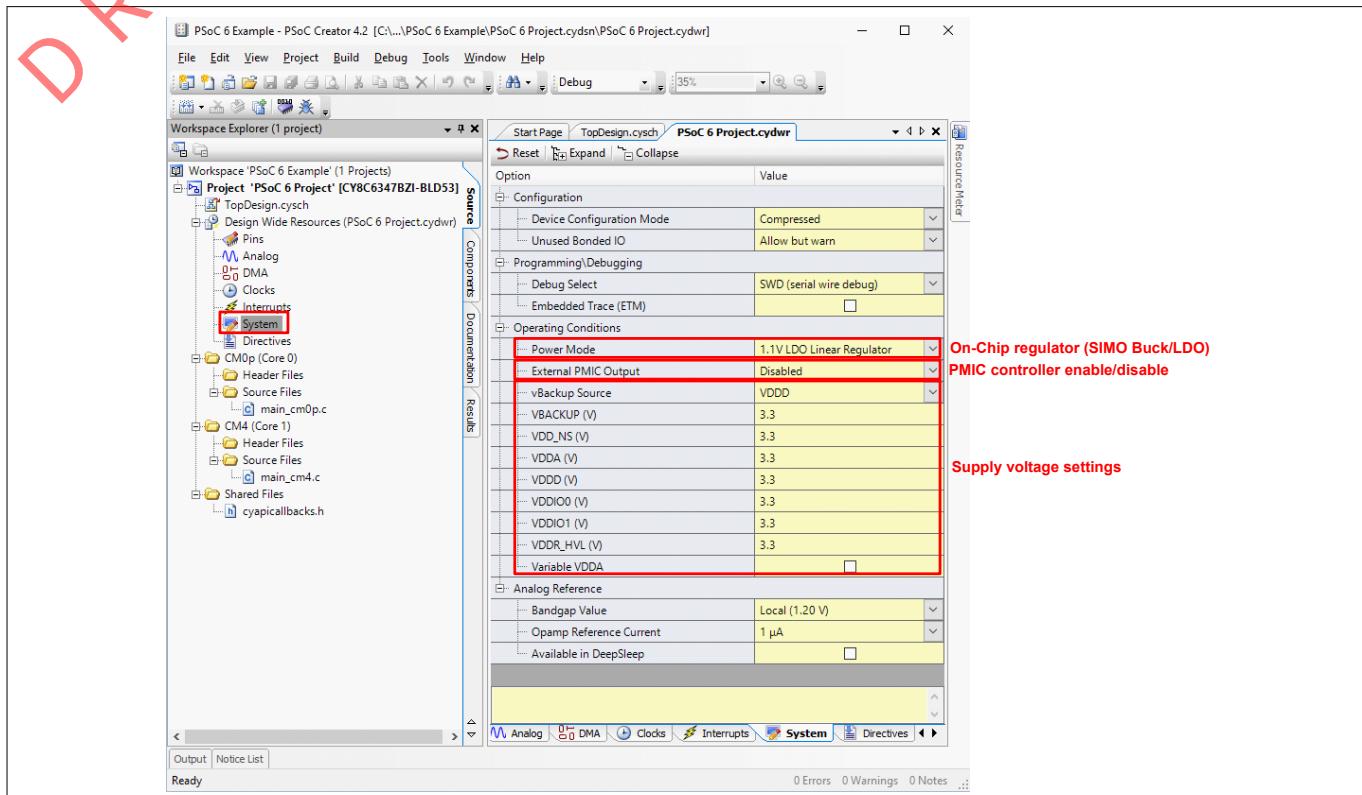


Figure 486 Device power settings in PSoC™ Creator

Figure 487 shows how to manage device power settings using the Device Configurator from ModusToolbox™ IDE.

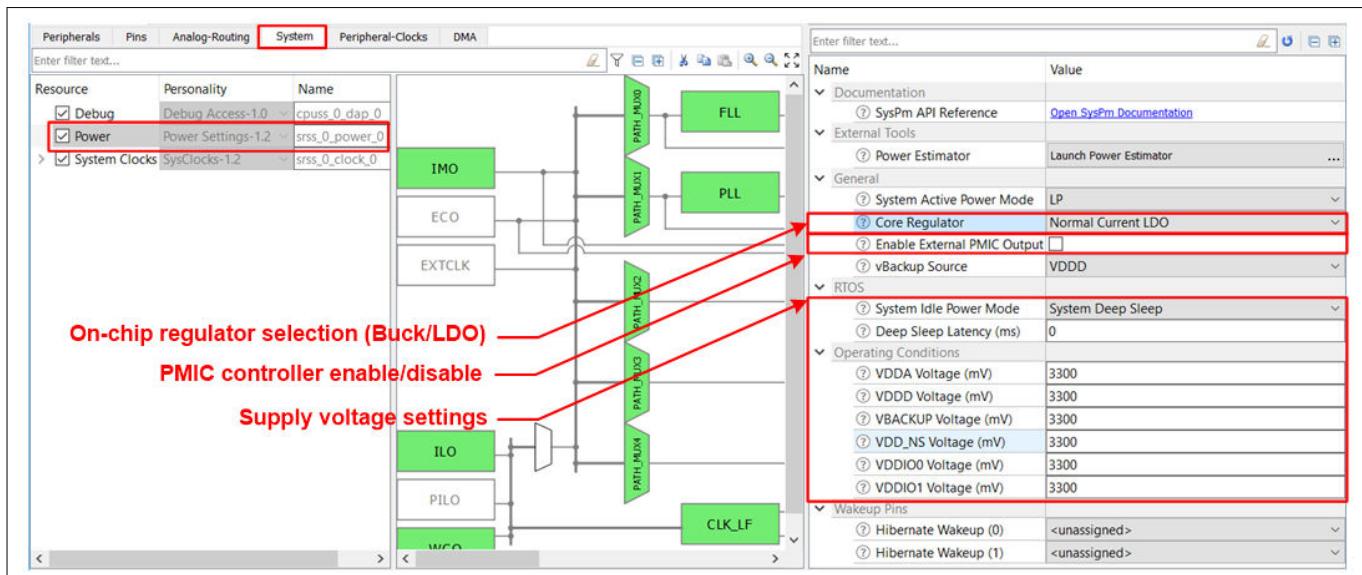


Figure 487 Device power settings in ModusToolbox™

5 PSoC™ 6 application notes~~DRAFT~~
5.13.3.6 Thermal considerations

Thermal considerations are important in the hardware design processes, such as package selection and PCB layout. PSoC™ 6 MCU targets low-power applications, as it consumes no more than 0.2 W. The maximum power consumption is so low that thermal considerations are not necessary.

5.13.3.7 eFuse programming

PSoC™ 6 MCU supports a 1024-bit One-Time-Programmable (OTP) eFuse. Each bit of the eFuse can be blown independently. See the "Nonvolatile Memory Programming" chapter of [PSoC™ 6 MCU: Architecture TRM](#) for details. While specific system calls are used to blow the eFuse bits, the VDDIO0 (or VDDIO if only one VDDIO is present in the package) supply of the device should be set to 2.5-V ($\pm 5\%$) for successfully programming/blowing the eFuse bits. The programming voltage of eFuse block is connected to V_{DDIO0} internally. Typically, eFuse programming is done only once before deployment. You can either blow the eFuse bits in the device before putting it on the hardware or make a provision to connect 2.5-V to VDDIO0 in the hardware for eFuse programming. eFuse programming is supported in PSoC™ Programmer 3.27 or later.

~~DO NOT USE~~ 5 PSoC™ 6 application notes

5.13.4 Clocking

The PSoC™ 6 MCU clocking system includes three internal clock sources: 8-MHz internal main oscillator (IMO), 32-kHz internal low-speed oscillator (ILO), and precision 32-kHz internal low-speed oscillator (PILO). Note that that PILO is not available in CY8C61x8/A, CY8C62x8/A, CY8C61x5, CY8C62x5, and CY8C62x4 devices. IMO has an accuracy of ± 2 percent across voltage and temperature. ILO has an accuracy of ± 10 percent. PILO has an accuracy of ± 2 percent, which can be calibrated to ± 500 ppm with a high-accuracy clock source.

Besides the internal clock sources, PSoC™ 6 MCU has three external clock sources: External clock (EXTCLK) generated using a signal from an I/O pin, external 16-35 MHz crystal oscillator (ECO), and external 32.768-kHz watch crystal oscillator (WCO).

The BLE radio in PSoC™ 6 MCU includes an additional crystal oscillator (32 MHz). An external 32 MHz crystal is mandatory for proper BLE operation. The clock generated by this oscillator block is available for other blocks inside PSoC™ 6 MCU. This clock is routed as the AltHF clock in the clocks settings discussed in [Clock settings](#). Note that this clock is available only when the BLE radio is powered. See [AN95089 - PSoC™ 4 BLE Crystal Oscillator Selection and Tuning Techniques](#) for details on selecting and tuning WCO/ECO crystals for BLE applications.

For more details, see the Clocking chapter of the [PSoC™ 6 MCU: Architecture TRM](#).

5.13.4.1 Clock settings

You can use either of the development environment to manage clocks.

Using PSoC™ Creator, you can configure sources and paths for High Frequency Clock (HFCLK) and Low Frequency Clock (LFCLK). The **Source Clocks** tab allows you to configure various clock sources and the **FLL/PLL** tab allows you to configure the (frequency-locked loop) FLL and phase-locked loop (PLL) inside PSoC™ 6 MCU. Switch to the **Clocks** tab in the DWR window, and double-click any row in the table of clocks or click the **Edit Clock** button to open the **Configure System Clocks dialog**, as Figure 488 shows.

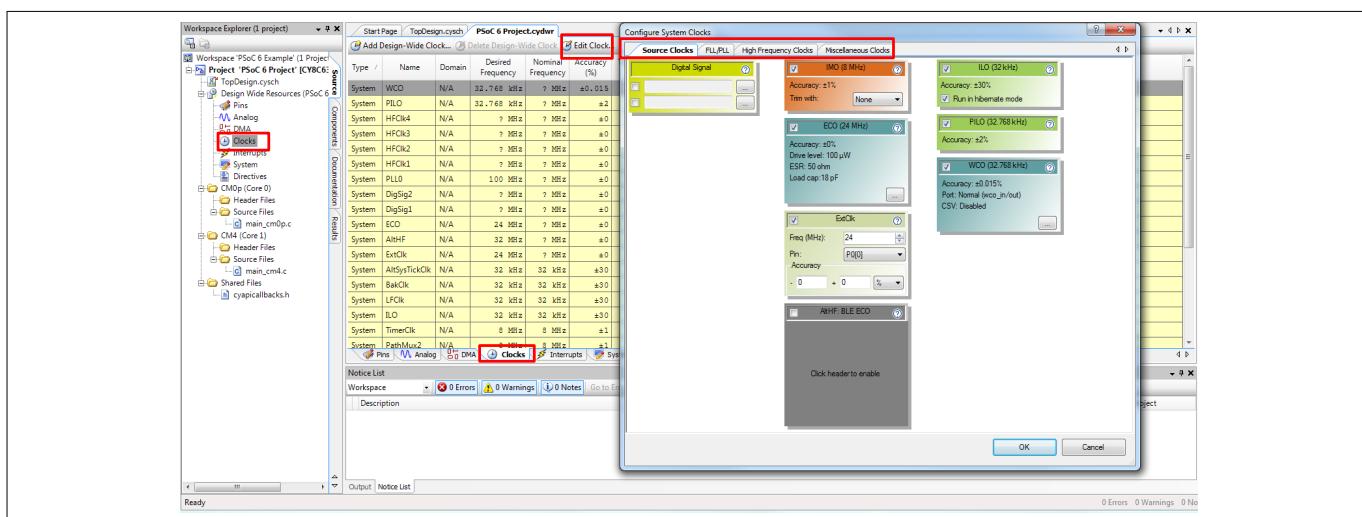


Figure 488 Clock settings in PSoC™ Creator

In ModusToolbox™ IDE, the **System** tab in the **Device Configurator** (design.modus) provides the options for configuring the clocks. Figure 489 shows how to configure system clocks using ModusToolbox™.

5 PSoC™ 6 application notes

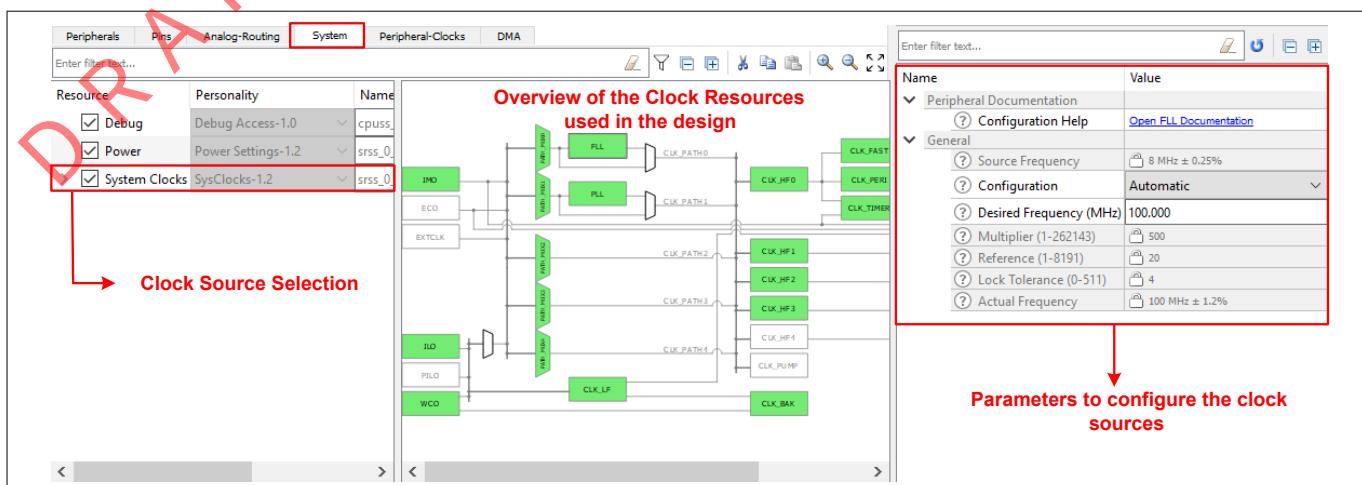


Figure 489 Clock settings in ModusToolbox™

5.13.4.1.1 Crystal oscillators

ECO

The ECO block requires an external crystal (16 MHz to 35 MHz) connection to appropriate pins as shown in Figure 490.

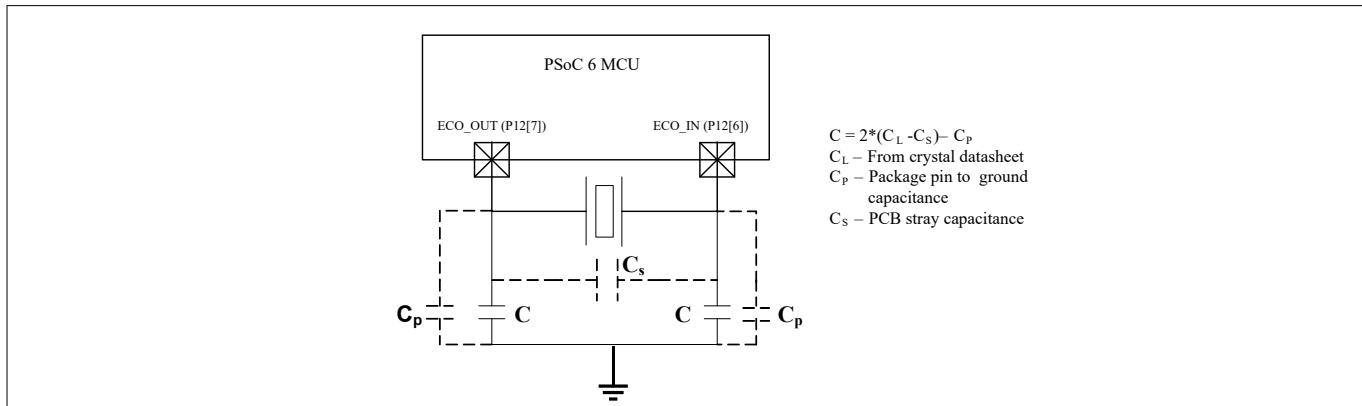


Figure 490 ECO connections

In PSoC™ 6 MCU, the external crystal connects to Port 12[6] and 12[7]. When the ECO is enabled, the corresponding I/O pins are configured appropriately by both PSoC™ Creator and ModusToolbox™ IDE to interface the external crystal.

Crystal manufacturers typically provide numerical values for parameters, namely the maximum drive level (DL), the equivalent series resistance (ESR), and the parallel load capacitance (C_L). These data should be entered in the Configure ECO settings to configure the ECO appropriately. The external load capacitors for the ECO are calculated as:

$$C = 2 \times (C_L - C_S) - C_P$$

where C_L – Crystal load capacitor as per the crystal datasheet

C_S – PCB stray capacitance. A well-designed PCB to minimize the stray capacitance includes a grounded copper wire between the crystal input and output wires.

C_P – Package pin to ground parasitic capacitance (typical value of 3 pF. See the device [datasheet](#) for more details on pin parasitic capacitance).

5 PSoC™ 6 application notes

It should be noted that the ECO available as part of the BLE radio has its own pins (XI and XO). You can add the load capacitor, crystal accuracy, and startup time details in the AltHF clock configure window as shown in Figure 491. External load capacitors for the BLE ECO are not required.

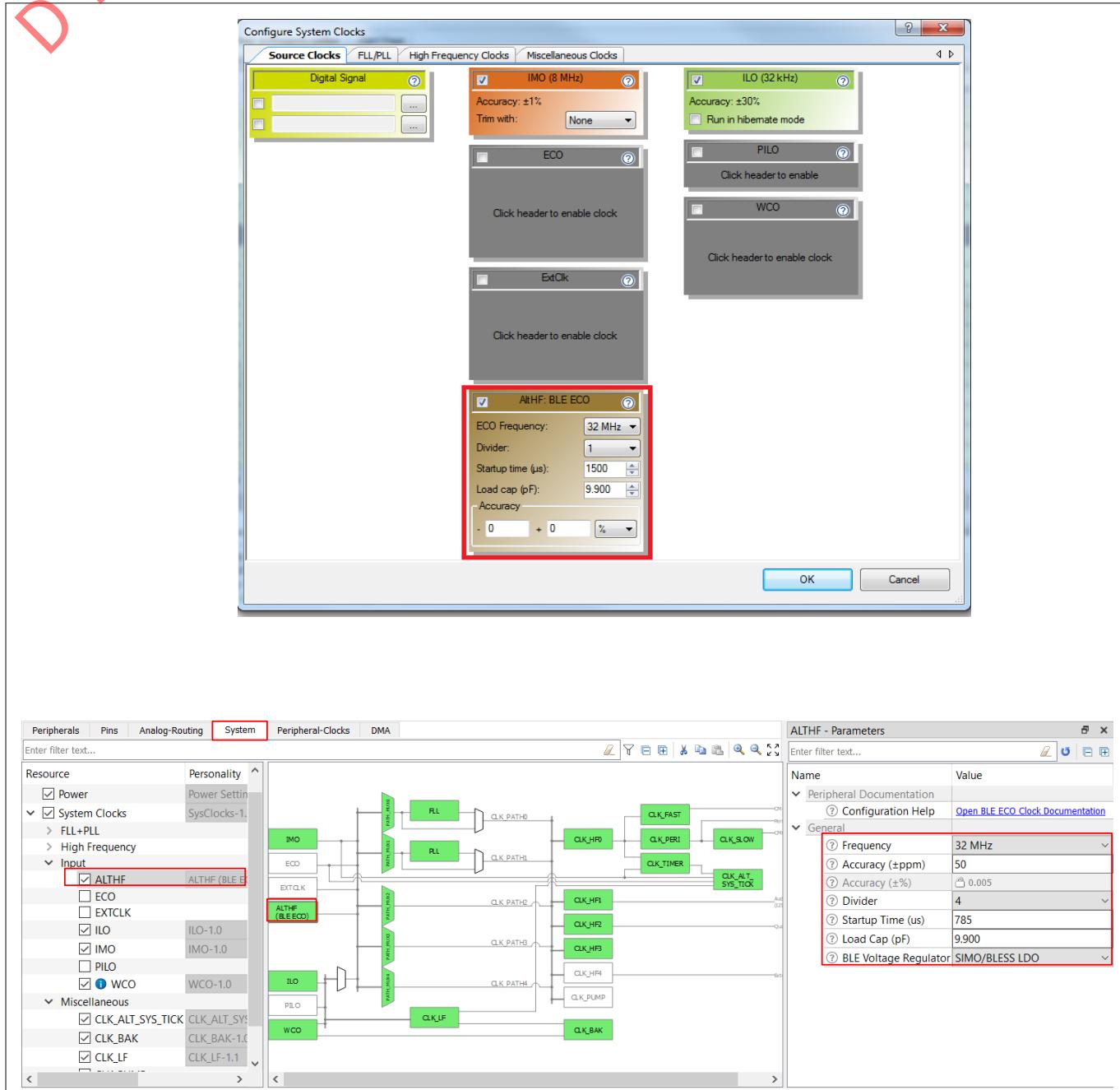


Figure 491 Configuring BLE ECO in PSoC™ Creator and ModusToolbox™

WCO

The WCO block requires an external 32.768-kHz crystal along with input and output load capacitors for proper operation. This is shown in Figure 492.

5 PSoC™ 6 application notes

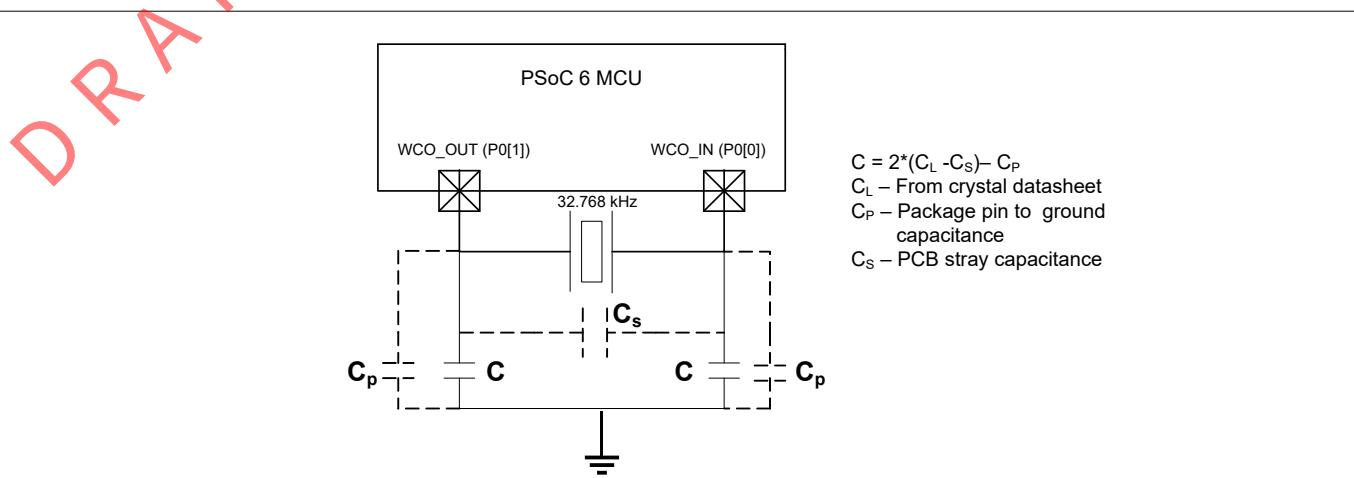


Figure 492 **WCO connections**

In addition, the WCO block can be bypassed altogether and a 32.768-kHz external sine wave can be directly fed to the WCO_OUT pin. In this configuration, the WCO_IN pin should be left floating. To bypass the WCO using PSoC™ Creator, enable WCO in **Configure System clocks** window and set the **Clock port** radio button in the **Configure WCO** dialog box to **Bypass (External sine wave)** as shown in [Figure 493](#). In ModusToolbox™, enable WCO in the **Device Configurator** (design.modus) and set the Clock Port to **Bypass (External sine wave)** as shown in [Figure 494](#). This will configure the WCO block to route a 32.768-kHz clock (sine wave signal) on WCO_OUT pin to RTC and LFCLK. Make sure that the WCO_IN pin is not used and left floating in the design.

The external load capacitors for the WCO are calculated as:

$$C = 2 \times (C_L - C_S) - C_P$$

where C_L – Crystal load capacitor as per the crystal datasheet

C_S – PCB stray capacitance. A well-designed PCB to minimize the stray capacitance includes a grounded copper wire between the crystal input and output wires.

C_P – Package pin to ground parasitic capacitance (typical value of 3 pF. See the device [datasheet](#) for more details on pin parasitic capacitance).

5 PSoC™ 6 application notes

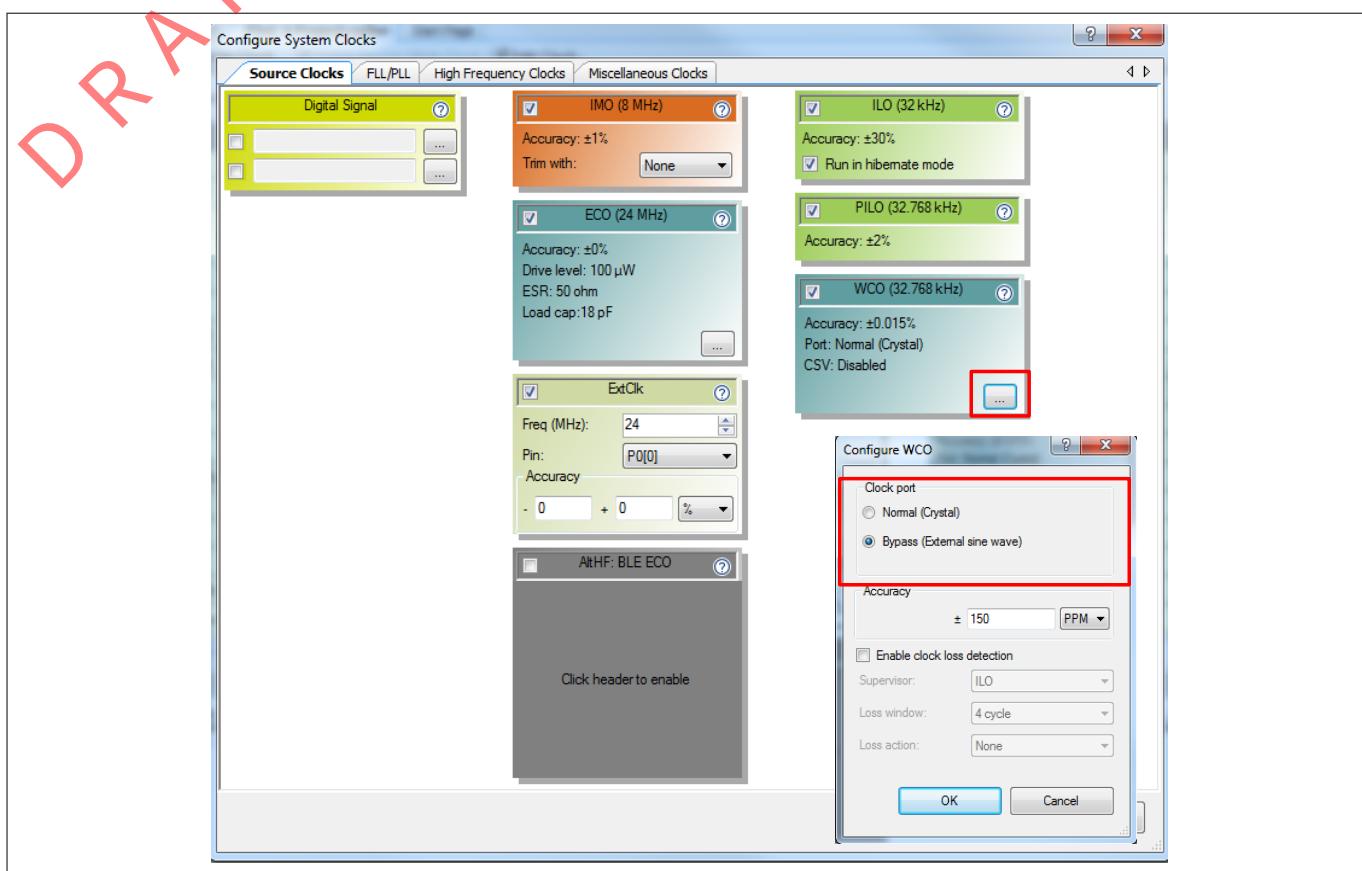


Figure 493 Configure WCO option in PSoC™ Creator

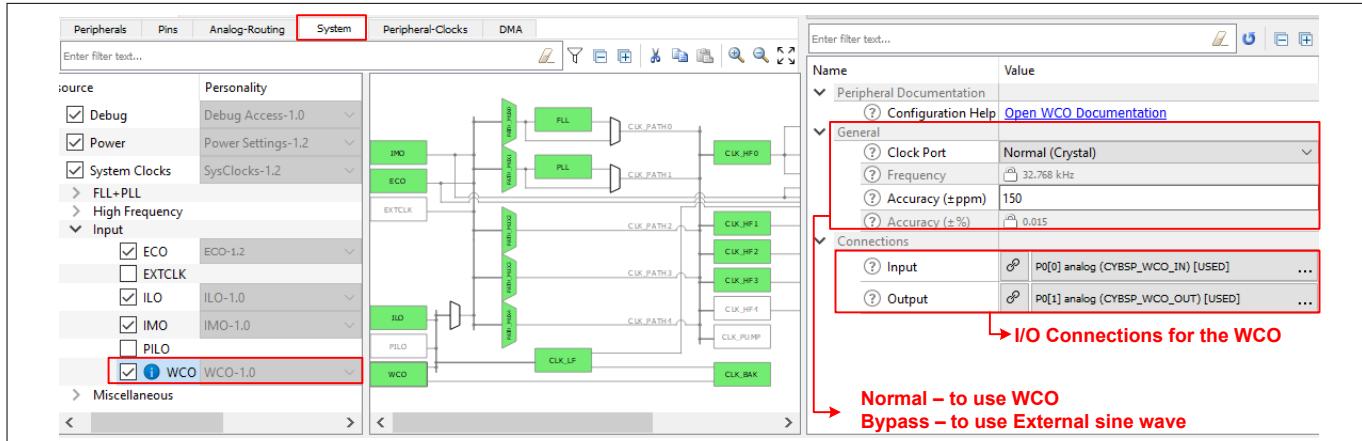


Figure 494 Configure WCO in ModusToolbox™

5.13.4.1.2 External clock

In PSoC™ 6 MCU, a 0–100 MHz-range clock can be connected to an EXT_CLK pin (P0[0] or P0[5]) and routed to various blocks inside PSoC™. This is shown in [Figure 495](#) and [Figure 496](#). When a pin is selected for receiving the external clock, it is automatically configured and reserved for EXTCLK by both PSoC™ Creator and ModusToolbox™. PSoC™ expects a 0–100 MHz-range digital signal with a 45–55% duty cycle on the EXTCLK input.

5 PSoC™ 6 application notes

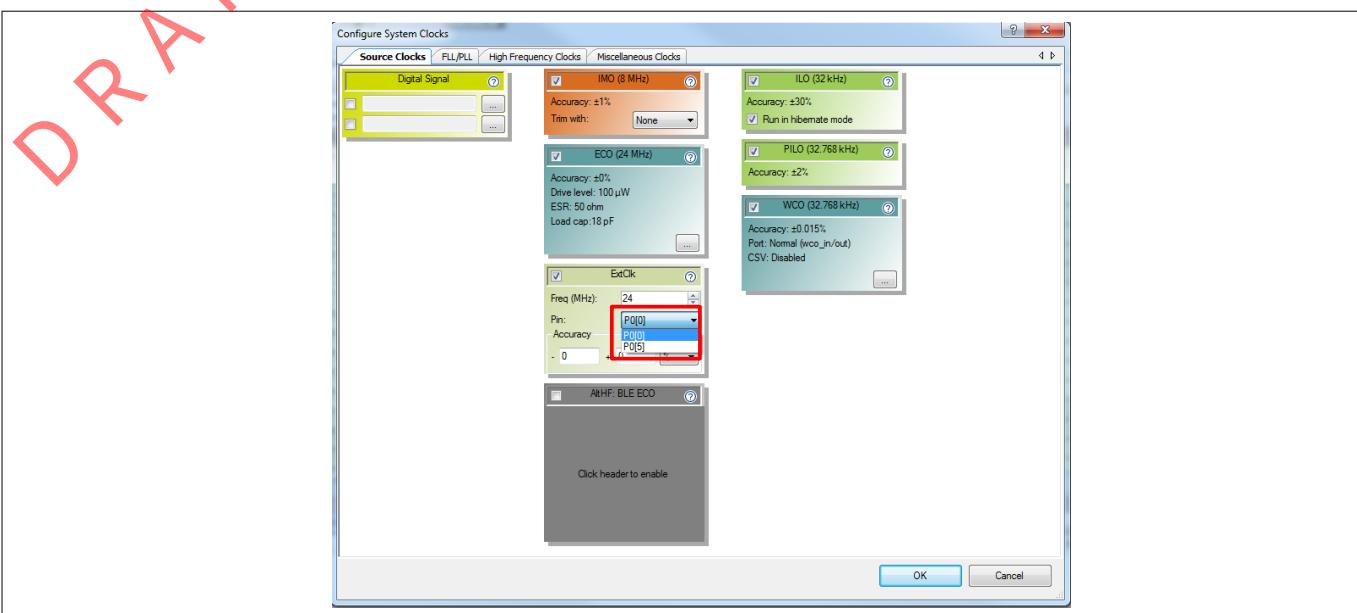


Figure 495 Routing external clock in PSoC™ Creator

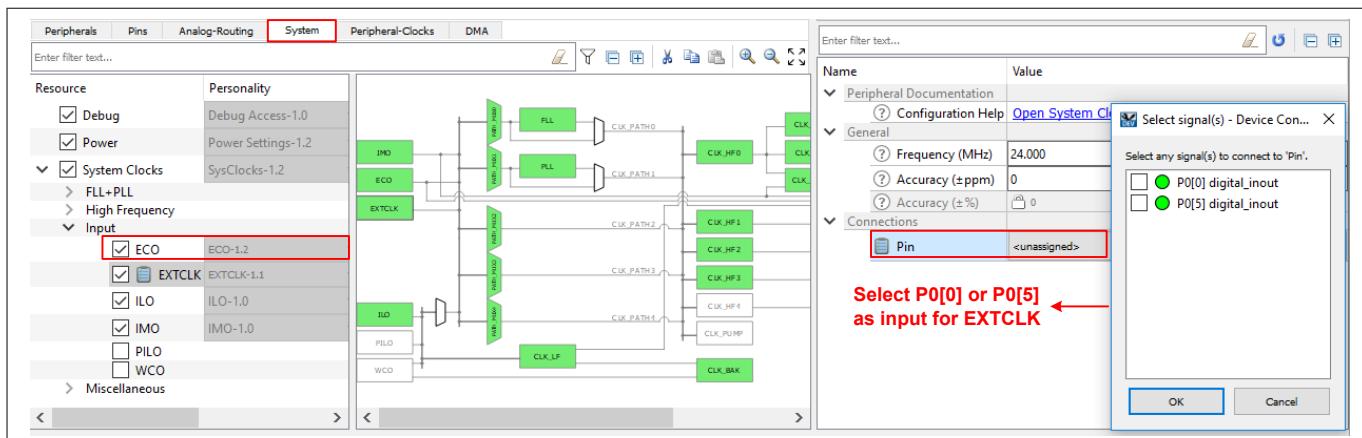


Figure 496 Routing external clock in ModusToolbox™

In addition, the output of CLK_HF4 can be routed out through P0[0] or P0[5]. Note that both the EXTCLK input and CLK_HF4 output use the same signal path. Hence only one of them can be active at a time. To use P0[0] or P0[5] as EXTCLK input, the HSIOM setting should be set to SRSS_EXT_CLK and drive mode should be configured as high impedance digital with input buffer enabled. To use the pin as CLK_HF4 output, the HSIOM setting should be set to SRSS_EXT_CLK and drive mode should be configured as strong drive with input buffer disabled.

5 PSoC™ 6 application notes

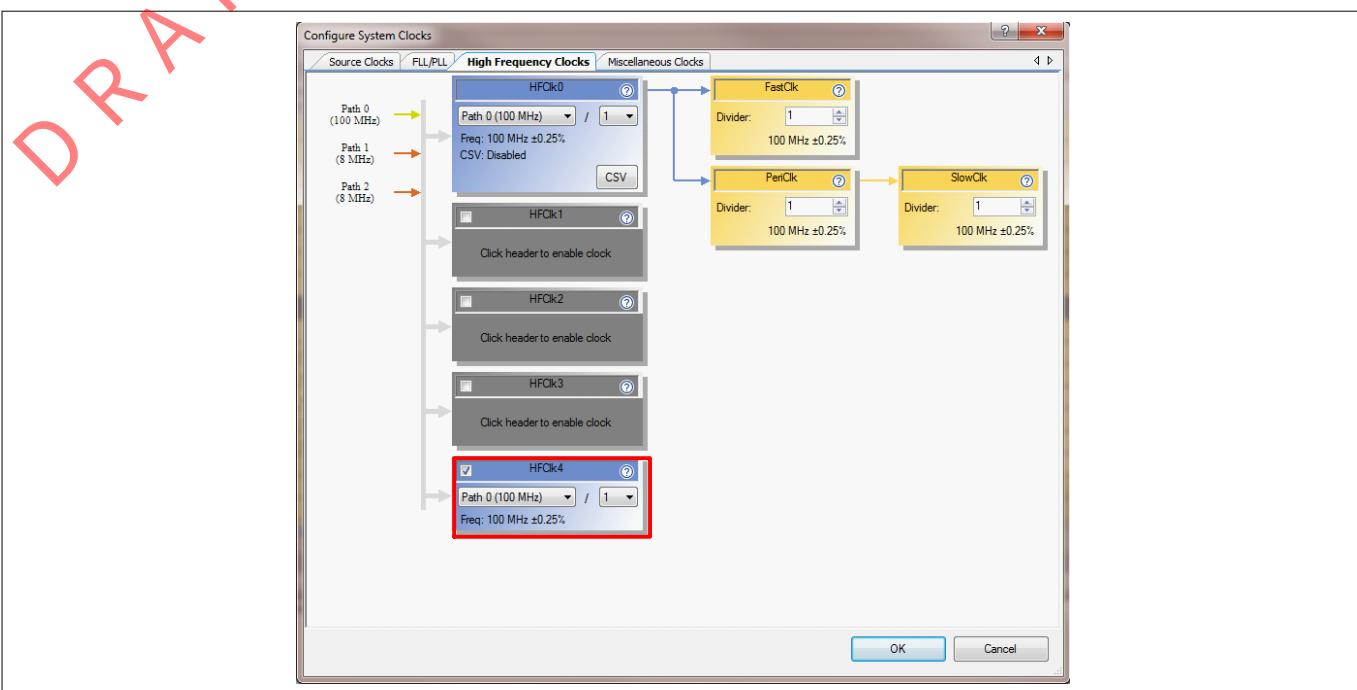


Figure 497 CLK_HF4 option in PSoC™ Creator

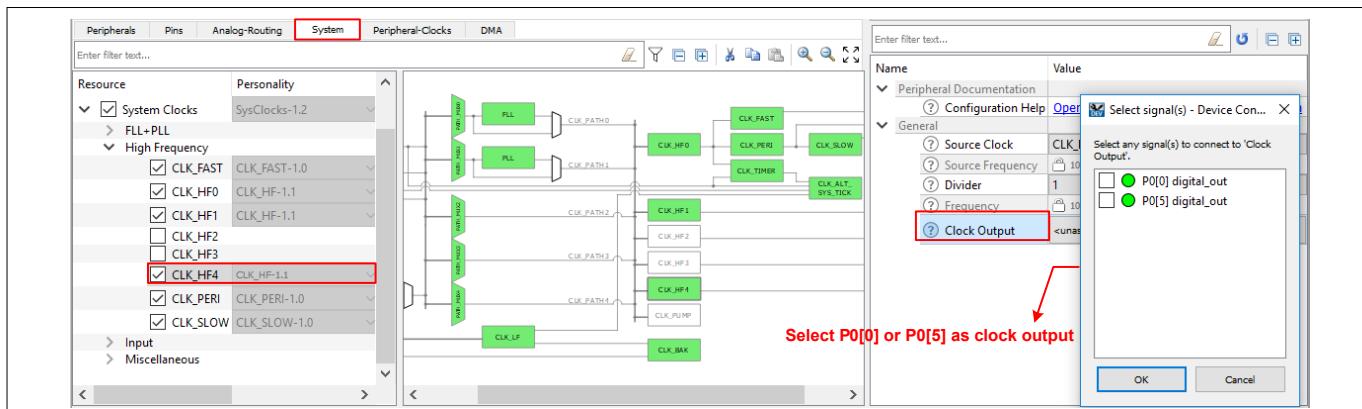


Figure 498 CLK_HF4 option in ModusToolbox™

~~DO NOT USE~~
5 PSoC™ 6 application notes**5.13.5 Reset**

PSoC™ 6 MCU has a reset pin, XRES, which is active LOW. You have to externally pull up the XRES pin to VDDD via a $4.7\text{ k}\Omega$ resistor. This will make sure that the XRES pin is not left floating in the design and the device can function properly. You can also connect a capacitor (typically $0.1\text{ }\mu\text{F}$) to the XRES pin, as [Figure 499](#) shows, to filter out glitches and give the reset signal better noise immunity. Optionally, if PSoC™ is controlled by an external host, the XRES pin can be directly driven by the host.

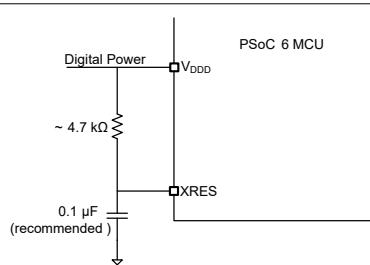


Figure 499 XRES pin connection

~~5 PSoC™ 6 application notes~~

~~5.13.6~~ Programming and debugging

The PSoC™ MCU Program and Debug interface provides a communication gateway for an external device to perform programming or debugging. The external device can be a Infineon-supplied programmer and debugger or a third-party device that supports programming and debugging. The serial wire debug (SWD) or the JTAG interface can be used as the program/debug protocol between the external device and PSoC™ 6 MCU. In addition, PSoC™ 6 MCU supports Arm™ Embedded Trace Macrocell (ETM) on the Cortex®-M4 CPU.

5.13.6.1 SWD

For SWD programming or debugging, you can use the onboard programmer/debugger of PSoC™ 6 MCU Kits (KitProg), or connect PSoC™ BLE 6 to an external debugger such as [CY8CKIT-002 MiniProg3](#) via any connector supported by the debugger. MiniProg3 supports a 10-pin and a 5-pin connector for SWD programming and debugging (see [Figure 500](#)). In addition to SWD, PSoC™ 6 MCU supports single-wire viewer (SWV) interface defined by Arm™. The SWV interface is used for program and data monitoring, where the firmware may output data in a method similar to “printf” debugging on PCs, using a single pin. The SWV support in MiniProg3 is available only on the 10-pin header (see [Figure 501](#)).

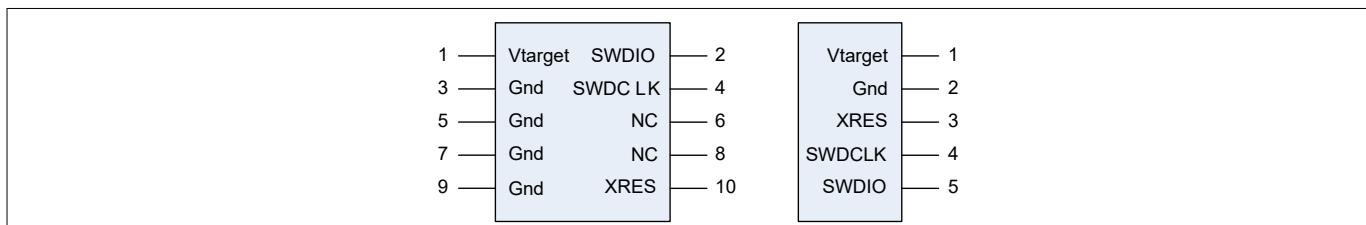


Figure 500 SWD connector pin maps for MiniProg3

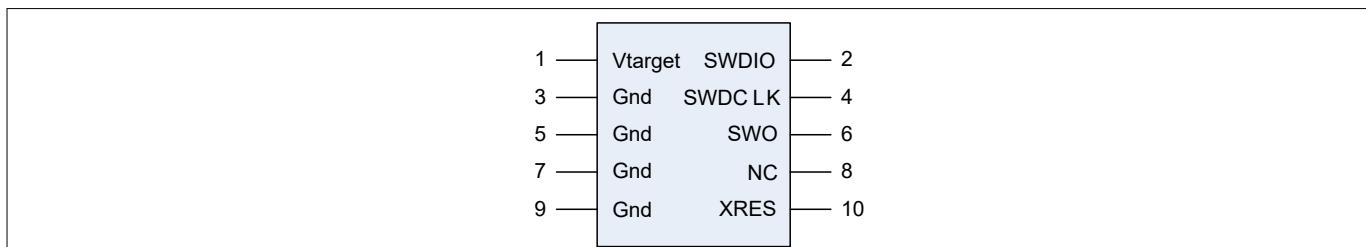


Figure 501 SWD+SWV connector pin maps for MiniProg3

[Figure 502](#) shows the SWD and SWV connections in PSoC™ 6 MCU.

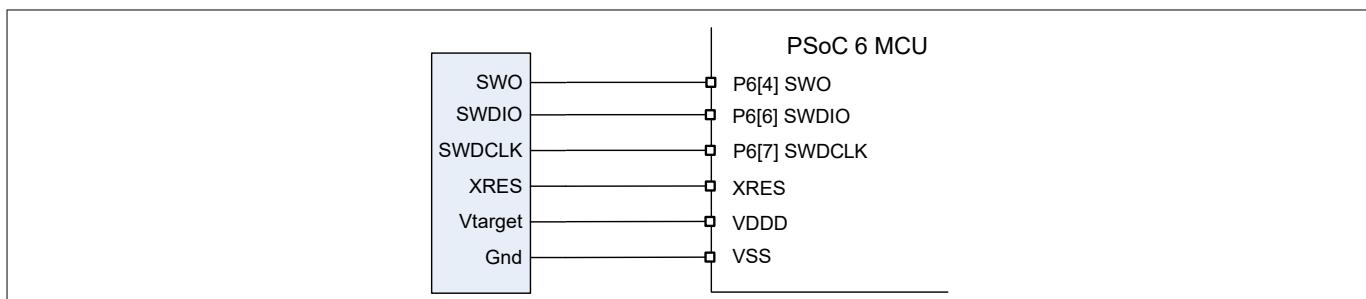


Figure 502 SWD/SWV connections to PSoC™ 6 MCU

5.13.6.2 JTAG

For JTAG programming and debugging, external debuggers like MiniProg3 or ULINK can be used. Both 4-wire and 5-wire JTAG programming is supported in PSoC™ 6 MCU. [Figure 503](#) shows the JTAG connections to PSoC™ 6 MCU. MiniProg3 supports 4-wire JTAG programming (see [Figure 504](#)) on the 10-pin connector.

5 PSoC™ 6 application notes

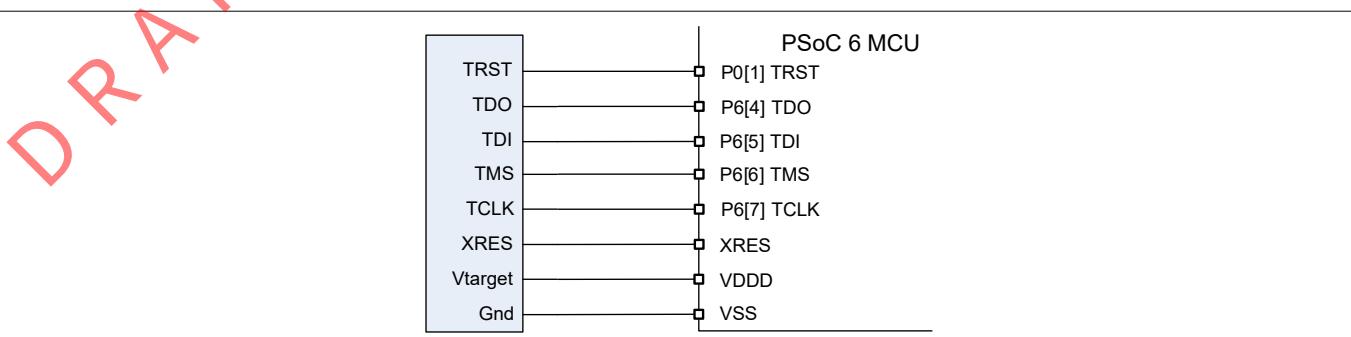


Figure 503 JTAG connections to PSoC™ 6 MCU

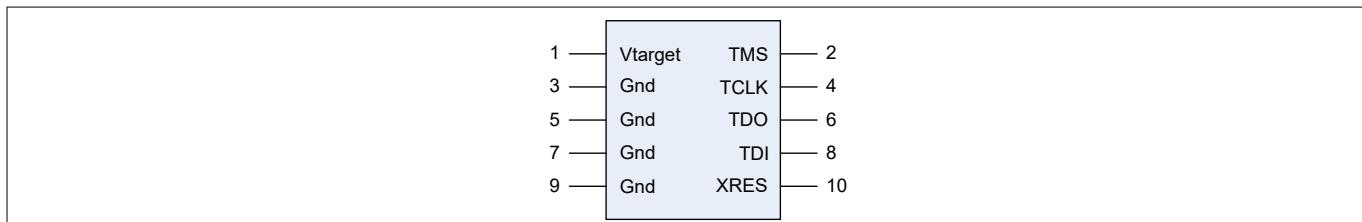


Figure 504 JTAG connections in MiniProg3 10-pin header

5.13.6.3 ETM

The Cortex®-M4 CPU in PSoC™ 6 MCU supports ETM. Any ETM trace viewers can be used with PSoC™ 6 MCU. ETM trace connections to PSoC™ 6 MCU are shown in [Figure 505](#).

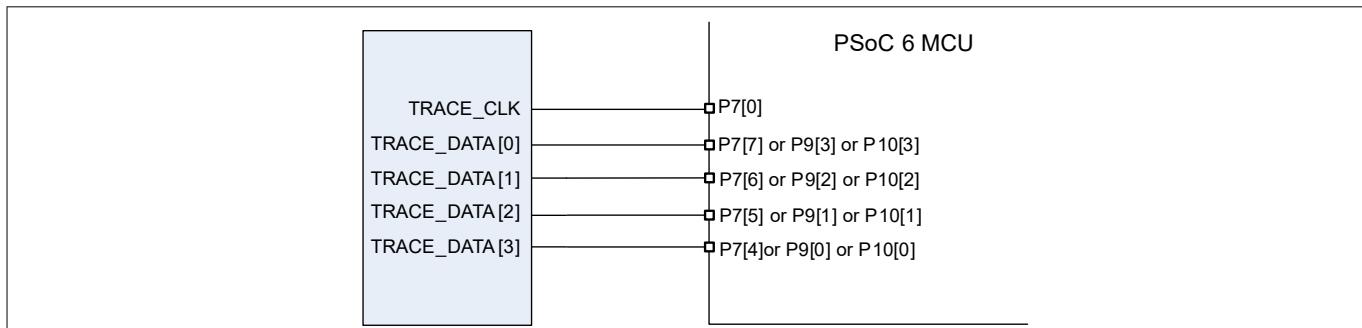


Figure 505 ETM connections to PSoC™ 6 MCU

5.13.6.4 Debug select

The SWD and JTAG pins could be used for other functionality when the devices are not being programmed; see the device [datasheet](#) for the possible functionality details. However, if you need to use SWD/JTAG pins for run-time debugging, select **SWD/4-wire JTAG/5-wire JTAG**, instead of **GPIO**, from the **Debug Select** pull-down list in the System tab of the DWR window, as [Figure 506](#) shows. [Figure 507](#) shows the corresponding debug settings in ModusToolbox™ IDE. In this case, the pins cannot be used for other functionality any longer. Similarly, if SWV and/or ETM trace is required, the appropriate option can be selected in the Debug Select dropdown (**SWD+SWV**) and checking the **Embedded Trace (ETM)** checkbox.

5 PSoC™ 6 application notes

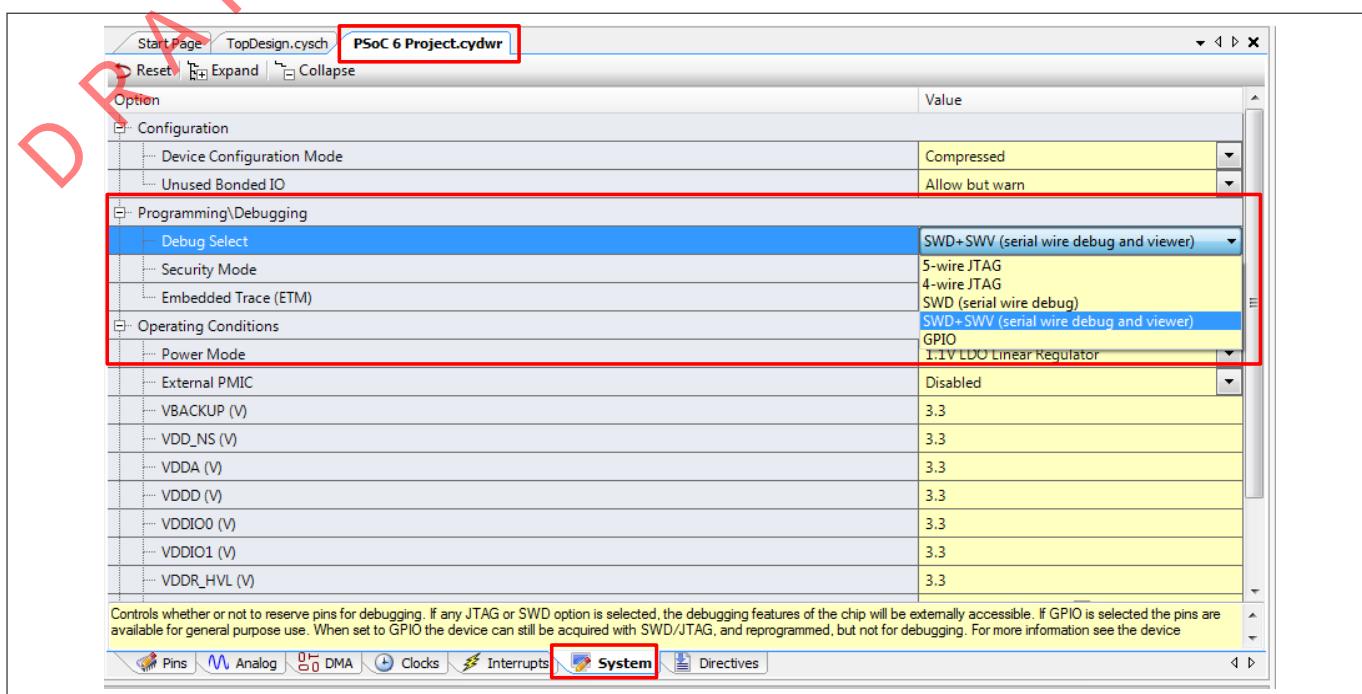


Figure 506 PSoC™ Creator debug settings

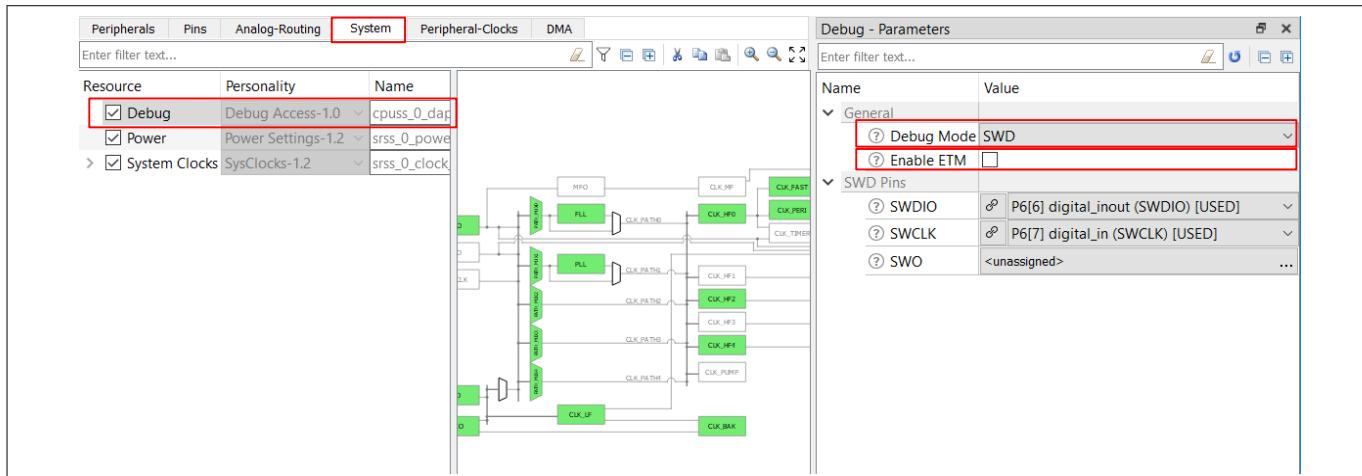


Figure 507 ModusToolbox™ debug settings

~~5 PSoC™ 6 application notes~~

~~5.13.7~~ **GPIO pins**

PSoC™ 6 MCU provides flexible GPIO pins. All GPIO pins can be controlled by firmware. Most of them also have alternative connections to PSoC™ 6 MCU peripherals. Different peripherals have different dedicated or fixed pins for their terminals. You get the best performance when a peripheral is connected to its own dedicated pin or pins. However, for flexibility, you can connect the peripheral to other pins at the cost of using some internal routing resources. The flexibility of PSoC™ devices and the capability of its I/O to route most signals to most pins greatly simplify circuit design and board layout. If a peripheral has fixed pins, then you can connect it only to those pins.

5.13.7.1 I/O pin selection

When you design a hardware system, based on PSoC™ 6 MCU, you should assign the GPIO pins in the following sequence as shown in [Table 102](#). See the device [datasheet](#) to determine whether the peripheral block listed in [Table 102](#) is supported by your PSoC™ 6 MCU device.

Table 102 I/O pin selection guide

Block	Pin name	Port#[Pin#]	Fixed/ dedicated	Remarks
System function pins				
Run-time Debug			Fixed	If you need run-time debugging, trace, or SWV support, select the appropriate setting in the System settings explained in Debug select . The selection will automatically lock the required IOs for the purpose.
External Crystal Oscillator (ECO)	ECO_IN	P12[6]	Fixed	External crystal frequency range: (16 MHz – 35 MHz); Use this ECO when BLE ECO is not used or the required crystal frequency is not 16/32 MHz.
	ECO_OUT	P12[7]	Fixed	
Watch Crystal Oscillator (WCO)	WCO_IN	P0[0]	Fixed	If you need a highly accurate and low-frequency clock for RTC or Deep Sleep wakeup purpose, use the WCO block with an external 32.768-kHz crystal or clock. Note that the WCO block is present in the device's backup domain and is available, even when the VDDD of the device is removed (VBACKUP supply should be present).
	WCO_OUT	P0[1]	Fixed	
Wakeup (Hibernate and PMIC controller)	HIB_WAKEUP	P0[4] or P1[4]	Fixed	The hibernate wakeup pin is used to wake PSoC™ 6 MCU from the hibernate mode. To wake up PMICs that supply V _{DDD} , use the WAKEUP_OUT pin. The PMIC wakeup signal can be generated from internal RTC alarms or an input on P0[4] (WAKEUP_IN).
	WAKEUP_OUT	P0[5]	Fixed	
	WAKEUP_IN	P0[4]	Fixed	

(table continues...)

5 PSoC™ 6 application notes

Table 102 (continued) I/O pin selection guide

Block	Pin name	Port#[Pin#]	Fixed/ dedicated	Remarks
External Clock	EXT_CLK	P0[0] or P0[5]	Fixed	<p>Configure the pin as input (high impedance digital) for receiving the external clock.</p> <p>Configure the pin as output (strong drive with input buffer disabled) for routing internal clock (CLK_HF4) out.</p>

Analog pins

Low-Power Comparator	LPCOMP.IN_P	P5[6], P6[2]	Dedicated	PSoC™ 6 MCU has two low-power comparators that can work in all system power modes.
	LPCOMP.IN_N	P5[7], P6[3]	Dedicated	
Opamp	Vplus Input	P9[0], P9[6], P9[5], P9[7]	Dedicated	PSoC™ 6 MCU has up to two opamps. These opamps are operational in Deep Sleep power mode and can be used for buffering, pre-amplifier, voltage follower and sample and hold operations by peripherals like SAR ADC and CT DAC. Note that this feature is not available in CY8C61x8/A, CY8C62x8/A, CY8C61x5, and CY8C62x5 devices. Note that the availability of pins depends on the selected PSoC™ 6 MCU device. See the respective device datasheet for the pin availability. For more details, see the Continuous Time Block mini (CTBm) chapter in PSoC™ 6 MCU Architecture TRM .
	Vminus Input	P9[1], P9[4]	Dedicated	
	Output	P9[2], P9[3]	Dedicated	
CAPSENSE™	CMOD	P7[1] or P7[2] or P7[7]	Fixed	For the self-capacitance method, connect a modulator capacitor to the CMOD pin and reservoir capacitor to the CSH_TANK pin.
	CSH_TANK	P7[1] or P7[2] or P7[7]	Fixed	
	CINT1	P7[1]	Fixed	For the mutual-capacitance method, connect two integrating capacitors CINT1 and CINT2 pins. See the CAPSENSE™ sections for details.
	CINT2	P7[2]	Fixed	
SAR ADC	SAR ADC pins	P10[0]-P10[7]	Dedicated	Port 10 has the dedicated connection to SAR ADC. You can route the ADC connection to other ports using AMUX A and AMUX B. Port 9 is the preferred port after port 10 because connection to other ports involve additional switch resistance in their paths.

Digital pins

(table continues...)

5 PSoC™ 6 application notes

DRAFT
Table 102 (continued) I/O pin selection guide

Block	Pin name	Port#[Pin#]	Fixed/ dedicated	Remarks
Timer/Counter Pulse-Width Modulator (TCPWM)	TCPWM pins	See the device datasheet	Dedicated	PSoC™ 6 MCU has up to 32 TCPWM blocks with each block having two complimentary PWM signals. All these signals are routed to dedicated GPIO pins.
Serial Communication Block (SCB)	SCB pins	See the device datasheet	Fixed	CY8C61x6/7, CY8C62x6/7, and CY8C63x6/7 devices have up to 9 SCBs out of which 8 SCBs can be configured as SPI, I ² C, or UART. One SCB supports only I ² C or SPI mode and is available in Deep Sleep power mode. CY8C62x8/A devices have up to 13 SCBs out of which 8 SCBs can be configured as SPI, I ² C, or UART, four SCBs can be either UART or I ² C, and one SCB is operational in Deep Sleep supporting I ² C or SPI mode. CY8C62x4, CY8C61x5, and CY8C62x5 devices have 7 SCBs out of which 6 SCBs can be configured as SPI, I ² C, or UART and one SCB is operational in Deep Sleep supporting I ² C or SPI mode.
Serial Memory Interface (SMIF)	SMIF pins	P11[0]-P11[7] P12[0]-P12[4]	Fixed	The SMIF block uses fixed pins. See the SMIF section and device datasheet for details on these pins.
Secure Data Host Controller (SDHC)	SDHC pins	See the device datasheet	Fixed	The SDHC block uses fixed pins. See the SDHC section and device datasheet for details on these pins.
Controller Area Network (CAN FD)	CAN pins	P5[0], P5[1]	Fixed	CAN with Flexible Data rate (CAN FD). CAN FD block is available in CY8C61x5, CY8C62x5, and CY8C62x4 devices only. See the device datasheet .
Segment LCD	Com and Seg pins	Routing is available to almost all the GPIO pins	-	Segment LCD drive block is available in CY8C61x5, CY8C62x5, and CY8C62x4 devices only. This block can operate in Deep Sleep power mode. See the device datasheet .
Audio Block	PDM	PDM_DATA PDM_CLK	See the device datasheet	Fixed The Audio subsystem consists of an I ² S block and two PDM channels. CY8C62xA devices have two I ² S blocks and two PDM channels See the Audio Subsystem section and device datasheet for details on these pins

(table continues...)

5 PSoC™ 6 application notes**Table 102 (continued) I/O pin selection guide**

Block	Pin name	Port#[Pin#]	Fixed/ dedicated	Remarks
I2S	I2S_TX_SCK I2S_TX_WS I2S_TX_SDO I2S_RX_SCK I2S_RX_WS I2S_RX_SDO I2S_MCLK	See the device datasheet	Fixed	

Note: For some devices in the PSoC™ 6 MCU family, simultaneous GPIO switching with unrestricted drive strengths and frequency can induce noise in on-chip subsystems affecting CAPSENSE™ and ADC results. For more details, see the Errata section of the corresponding device [datasheet](#).

5 PSoC™ 6 application notes

5.13.8 Analog module design tips

5.13.8.1 CAPSENSE™

In the self-capacitance mode, you can connect any PSoC™ 6 MCU pin to a CAPSENSE™ sensor except **CMOD** (or **C_MOD**) pin, which is reserved for the modulating capacitor (C_{MOD}) function. In PSoC™ 6 MCU, CMOD should be connected to P7[1] or P7[2] or P7[7]. When you need to use a shield electrode for waterproofing or proximity features, you may also need to reserve the **CTANK** (or **C_SH_TANK**) pin for the tank capacitor, C_{SH_TANK} . In PSoC™ 6 MCU, C_{SH_TANK} can be connected to P7[1] or P7[2] or P7[7]. If the parasitic capacitance of the shield is less than 200 pF, it is optional to use C_{SH_TANK} ; otherwise, it is recommended to use the tank capacitor for improved water tolerance. The value for C_{MOD} is usually 2.2 nF. The value of C_{SH_TANK} is usually 10 nF.

In mutual capacitance, you can connect any PSoC™ 6 MCU pin to a CAPSENSE™ Rx/Tx sensor. Two integrating capacitors (C_{INT1} and C_{INT2}) are required for proper operation. A 470-pF capacitor is recommended on C_{INT1} and C_{INT2} . In PSoC™ 6 MCU, C_{INT1} and C_{INT2} should be connected to P7[1] and P7[2].

CAPSENSE™ detects a finger touch by a tiny variation in the sensor's capacitance (less than 1 pF). It is very sensitive to both signal and noise. Note the PCB layout tips for CAPSENSE™. See [PSoC™ 4 and PSoC™ 6 MCU CAPSENSE™ Design Guide](#) for more details.

Pins with a large sink current that are close to CAPSENSE™ pins can introduce an offset to the CAPSENSE™ module's "GND." [Figure 508](#) illustrates a switch circuit for CAPSENSE™ in the IDAC source mode. R1 and R2 represent the resistances of PSoC™ 6 MCU internal traces, and R3 represents the resistance of a PCB trace. A shared return path of sink current and CAPSENSE™ current is composed of R2 and R3. The closer a pin is to the CAPSENSE™ pin, the higher is the offset generated with increase in sink current that flows through the return path.

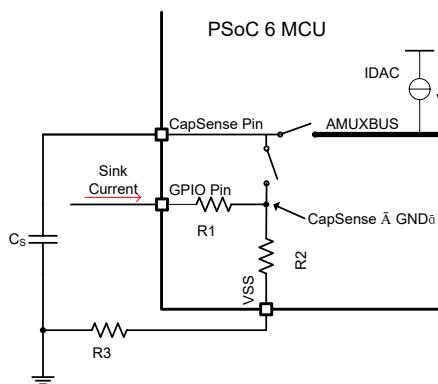


Figure 508 Sharing return path

This offset is undesirable and may cause fluctuations in the CAPSENSE™ reading and possible false triggers. Offset compensation can be done in firmware, but it is strongly recommended that you remove the offset in the hardware design instead. Keep pins with a large sink current as far as possible from the CAPSENSE™ pins (best practice is by more than three pins). In addition, pay attention to the return path in your PCB. See [AN57821 – PSoC™ 3, PSoC™ 4, and PSoC™ 5LP Mixed-Signal Circuit Board Layout Considerations](#) for more details on mixed-signal circuit design.

The CAPSENSE™ block can be time shared for implementing both mutual and self-capacitance mode of sensing. For this, one has to connect the CMOD capacitor to P7[7] for CSD mode of sensing and CINT1 and CINT2 capacitors to P7[1]/P7[2] for CSX mode of sensing. This method, however, does not allow the use of a tank capacitor for CSD operation.

In PSoC™ 6 MCU, follow the pin preference shown in [Table 103](#) for routing the CAPSENSE™ sensor, shield, and Rx and Tx signals in your design. This preference is based on the pin's proximity to the CAPSENSE™ block inside PSoC™ 6 MCU. Choosing a lower preference port might result in slight performance degradation. If the board

5 PSoC™ 6 application notes

~~DRAFT~~
design/routing suits a lower-preference I/O port, it can be selected. Use this preference only if routing is not a constraint.

Table 103 CAPSENSE™ port preference for routing

Preference	CAPSENSE™ sensor	CAPSENSE™ shield	CAPSENSE™ Rx	CAPSENSE™ Tx
First Preference	Port 5, Port 6, Port 7, Port 8			Any Port
Second Preference	Port 9 and Port 10			
Third Preference	Port 0, Port 1, Port 11, Port 12, and Port 13			

Note: *Simultaneous GPIO switching with unrestricted drive strengths and frequency can affect CAPSENSE™ performance. For selecting GPIO pins for CAPSENSE™ sensors, follow the recommendations given in AN85951 - PSoC™ 4 and PSoC™ 6 MCU CAPSENSE™ Design Guide.*

A quick CAPSENSE™ layout rule checklist is provided in Table 104.

Table 104 CAPSENSE™ layout quick guide

Category		Min	Max	Remarks/recommendations
Sensor Construction	Sensor Material	N/A	N/A	Copper, Indium Tin Oxide (ITO), printed ink on substrates such as glass, flex PCB
	Overlay Material	N/A	N/A	Needs to be non-conductive material with high permittivity: Glass, ABS Plastic, Formica Avoid using conductive paints on the overlay
	Widget			
	Button	Shape		Circle or rectangular with curved edges
		Size	5 mm	15 mm
		Clearance to ground hatch	0.5 mm	2 mm
		Overlay thickness	N/A	5 mm
	Slider	Width of segment	1.5 mm	8 mm
		Clearance between segments	0.5 mm	2 mm
		Height of segment	7 mm	15 mm
		Overlay thickness	N/A	5 mm

(table continues...)

5 PSoC™ 6 application notes

Table 104 (continued) CAPSENSE™ layout quick guide

Category			Min	Max	Remarks/recommendations
	Overlay Adhesive		N/A	N/A	Use a nonconductive adhesive film for bonding the overlay and the PCB. 3M™ makes a high-performance acrylic adhesive called 200MP that is widely used in CAPSENSE™ applications
PCB Layout	Sensor Traces	Width	N/A	7 mil	Use the minimum width possible with the PCB technology that you use
		Length	N/A	300 mm for a standard (FR4) PCB 50 mm for flex PCB	Keep as low as possible
		Clearance to ground and other traces	0.25 mm	N/A	Use maximum clearance while keeping the trace length as low as possible
		Routing	N/A	N/A	Route on the opposite side of the sensor layer. Isolate from other traces. If any non-CAPSENSE™ trace crosses the CAPSENSE™ trace, ensure that intersection is orthogonal. Do not use sharp turns
Via	Number of vias	1	2		At least one via is required to route the traces on the opposite side of the sensor layer
		Hole size	N/A	N/A	10 mil
Ground	Hatch fill percentage	N/A	N/A		Use hatch ground to reduce parasitic capacitance. Typical hatching: 25% on the top layer (7-mil line, 45-mil spacing) 17% on the bottom layer(7-mil line, 70- mil spacing)
Liquid Tolerance	Shield electrode	Spread	N/A	1 cm	If you have PCB space, use 1-cm spread
	Guard sensor	Shape	N/A	N/A	Rectangle with curved edges Recommended thickness of guard trace is 2 mm and distance of guard trace to shield electrode is 1 mm
EMC	Series resistor	Placement	N/A	N/A	Place the resistor within 10 mm of the PSoC™ pin

~~5 PSoC™ 6 application notes~~

~~5.13.8.2 SAR ADC~~

PSoC™ 6 MCU has a 12-bit differential SAR ADC (CY8C62x4 has two 12-bit SAR ADCs), with a sampling rate up to 2 Msps. Note that the maximum sampling rate of the SAR ADC depends on the selected PSoC™ 6 device. See the respective device [datasheet](#) to know the maximum SAR ADC sampling rate of the device. As mentioned in [I/O pin selection](#), SAR ADC uses dedicated GPIO pins for multichannel inputs. They provide the lowest parasitic path resistance and capacitance. You can also route the signals from other pins to the SAR ADC using the internal analog bus, but doing so will introduce high switch resistance (R_{SW} in [Figure 509](#)) and additional parasitic capacitance.

SAR ADC in PSoC™ 6 MCU has the following options for voltage reference:

- Internal V_{REF} (1.2 V or 0.8 V reference from AREF)
- V_{DDA}
- $V_{DDA}/2$
- External V_{REF}

PSoC™ 6 MCU also has an internal precision reference of 1.2 V (± 1 percent). You can use other internal references, including V_{DDA} and $V_{DDA}/2$, to extend the SAR ADC's input range. However, note that the accuracy of V_{DDA} and $V_{DDA}/2$ as references depends on your power system design, and it probably cannot be better than the 1.2-V precision reference. When you use the internal reference or $V_{DDA}/2$ as your reference, a bypass capacitor on the VREF pin can help you run the SAR ADC at a faster clock. See [Table 105](#) for details.

Table 105 References for SAR ADC

References	VDDA	Maximum SAR ADC clock frequency	Maximum sample rate	Supported devices
External reference	1.7 V – 3.6 V	18 MHz	1 Msps	All PSoC™ 6 devices
	2.7 V – 3.6 V	36 MHz	2 Msps	CY8C62x4, CY8C61x5, CY8C62x5 CY8C62x8/A, and CY8C61x8/A devices.
Internal reference without bypass capacitor	1.7 V – 3.6 V	1.8 MHz	200 ksps	CY8C62x4, CY8C61x5, CY8C62x5 CY8C62x8/A, and CY8C61x8/A devices.
			100 ksps	CY8C61x6/7, CY8C62x6/7 and CY8C63x6/7 devices
Internal reference with bypass capacitor	1.7 V – 3.6 V	18 MHz	1 Msps	All PSoC™ 6 devices
V_{DDA} as reference	1.7 V – 2.7 V	18 MHz	1 Msps	All PSoC™ 6 devices
	2.7 V – 3.6 V	36 MHz	2 Msps	CY8C62x4, CY8C61x5, CY8C62x5 CY8C62x8/A, and CY8C61x8/A devices.

If you need a reference with a higher accuracy or a specific voltage value, you can connect a custom external reference and a bypass capacitor to the VREF pin.

The SAR ADC is differential physically. When you select single-ended input mode, you must select the connection for the negative input. There are three options: VSS, VREF, and an external pin. The SAR ADC's input range is affected by the selection as well as by the value of the reference voltage. See the chapter "SAR ADC" in [PSoC™ 6 MCU Architecture TRM](#): devices for more information.

5 PSoC™ 6 application notes

DRAFT

Note: Simultaneous GPIO switching with unrestricted drive strengths and frequency can affect ADC performance. For more details, see the Errata section of the corresponding device [datasheet](#).

5.13.8.2.1 SAR ADC acquisition time

Another parameter of concern is the SAR ADC acquisition time, which depends on your hardware design, as [Figure 509](#) shows.

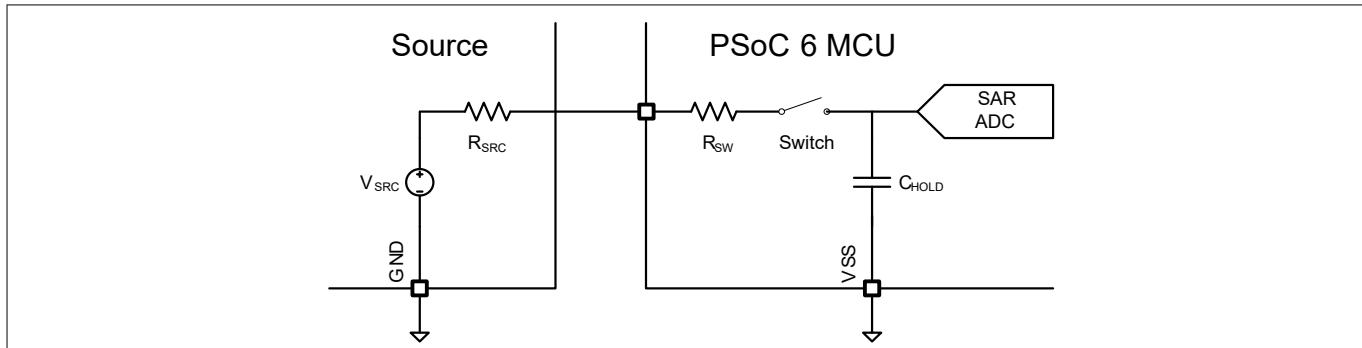


Figure 509 **Equivalent sample and hold circuit of PSoC™ 6 MCU SAR ADC**

V_{SRC} is the sampled signal source, and R_{SRC} is its output resistance. R_{SW} is the resistance of the path from a dedicated pin to the SAR ADC input. C_{HOLD} is the sample and hold capacitance. See the respective PSoC™ 6 MCU device [datasheet](#) for the values of R_{SW} and C_{HOLD} .

[Figure 510](#) shows how C_{HOLD} is charged during the acquisition time. During the acquisition time, the switch in [Figure 509](#) is on. Assuming that C_{HOLD} is charged from 0, the acquisition time is the time required to charge C_{HOLD} to a voltage level (V_{HOLD}) such that the error ($V_{SRC} - V_{HOLD}$) is less than the ADC's resolution.

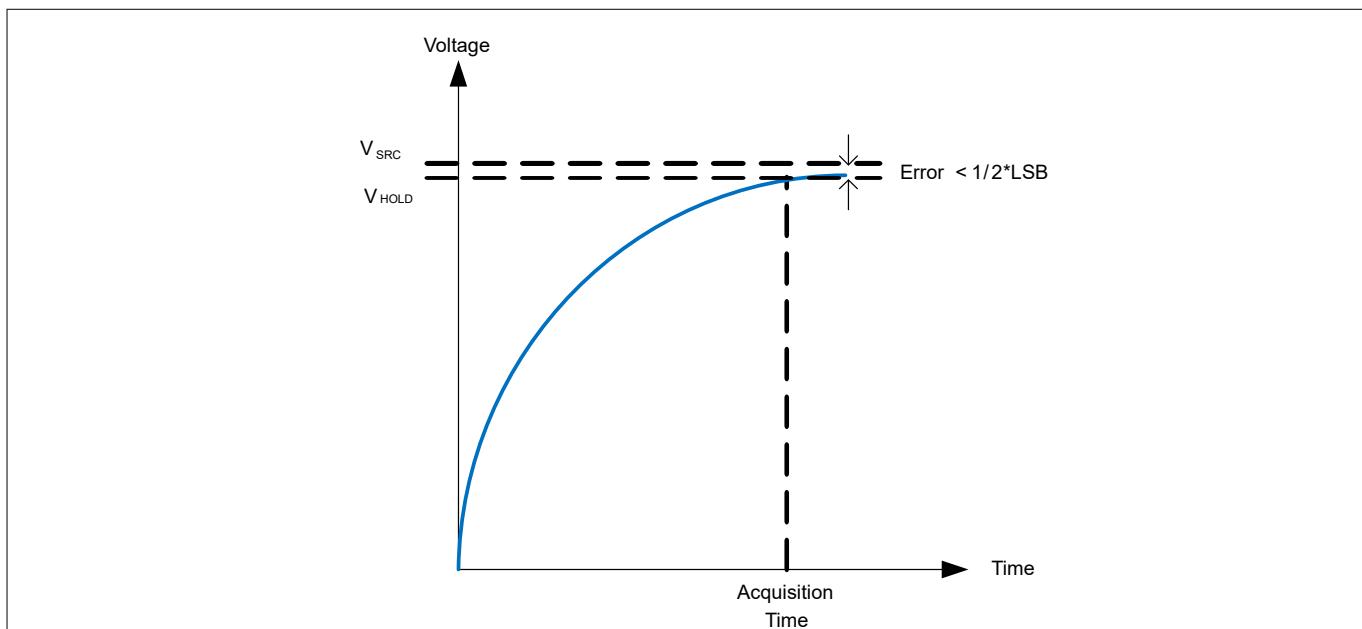


Figure 510 **C_{HOLD} charging process**

If the error is smaller than half the ADC's resolution ($1/2 * \text{LSB}$), it should be okay. The error can be related to the acquisition time in the following equation:

$$\text{Error} = V_{SRC} \times e^{-\frac{t_{ACQ}}{\tau}} = V_{SRC} \times e^{-\frac{t_{ACQ}}{(R_{SRC} + R_{SW}) \times C_{HOLD}}}$$

5 PSoC™ 6 application notes

~~DRAFT~~

Here, t_{ACQ} is the acquisition time, while [embedded OLE: embeddings/oleObject1.bin] is the charging time constant.

PSoC™ 6 MCU provides a 12-bit differential ADC. If V_{REF} is the reference voltage, the resolution can be expressed in the following equation:

$$LSB = \frac{2V_{REF}}{2^{12}}$$

This example assumes that the negative input is connected to V_{REF} , so that V_{SRC} has an input range from 0 to $2V_{REF}$. If the acquisition time is $9 * (R_{SRC} + R_{SW}) * C_{HOLD}$, the error can be expressed as follows:

$$Error = V_{SRC} \times e^{-9} \approx \frac{V_{SRC}}{8013} < \frac{2V_{REF}}{8013} \approx \frac{1}{2} \times \frac{2V_{REF}}{2^{12}} = \frac{1}{2} \times LSB$$

This equation shows that you should choose an acquisition time that is longer than $9 * (R_{SRC} + R_{SW}) * C_{HOLD}$ to make the error less than $1/2 * LSB$ of the 12-bit ADC.

In conclusion, pay attention to the output resistance of the sampled signal source, R_{SRC} , and the resistance introduced by PCB traces in your ADC hardware design. These determine the acquisition time and therefore the sampling rate. For example, in the CY8C62x4 device, the input resistance (R_{SW}) and input capacitance (C_{HOLD}) of the SAR ADC are 1 kΩ and 5 pF, respectively. If the output resistance of the sampled signal source is negligible, then the acquisition time of the SAR ADC can be calculated as $9 * (R_{SW}) * C_{HOLD}$, which is approximately 45 ns.

5.13.8.3 CTDAC

The PSoC™ 6 MCU analog subsystem supports a 12-bit continuous time digital-to-analog converter (CTDAC). The 12-bit DAC provides a continuous time output without the need for an external sample-and-hold (S/H) circuit. The CTDAC block can be used in applications that require voltage references, bias, or analog waveform output.

CTDAC can have one of the following sources as the input voltage reference:

- V_{DDA}
- Internal V_{REF}
- External voltage reference

The opamp OA1 in the CTBm block is configured as a voltage follower and used to route the external signal as the reference voltage to the CTDAC. The external signal is routed to the DAC using AMUX and therefore any GPIO can be used to connect the external signal. Note that any signal on the OA1 output terminal can be used as an external signal source if OA1 is disabled. In that case, the signal will be loaded by the DAC's resistive ladder directly. Therefore, the impact of such loading should be considered.

Note: CY8C61x8/A, CY8C62x8/A, CY8C61x5, and CY8C62x5 devices do not contain the CTDAC block.

The CTDAC output can be routed in three different paths. [Table 106](#) shows the CTDAC output ports.

Table 106 CTDAC output paths

CTDAC Output mode	Port#[Pin#]
Direct output path	P9[6]
Buffered output path via Opamp0	P9[2]
Sample and hold path using Opamp0 and C_{HOLD} capacitor	P9[2]

~~5 PSoC™ 6 application notes~~

~~5.13.9~~ Using external memory in the design

PSoC™ 6 MCU has a provision for interfacing with an external memory if your design requires it. The serial memory interface (SMIF) IP block is used for this purpose. The SMIF block primarily implements a single-SPI, dual-SPI, quad-SPI, or octal-SPI communication to interface with external memory chips. The SMIF block's primary use case is to set up the external memory and have it mapped to the PSoC™ memory space using the hardware. The SMIF block connects to dedicated pins. Therefore, if your design requires the use of an external memory, then the corresponding pins should be used. See the device [datasheet](#) for more details on the pins.

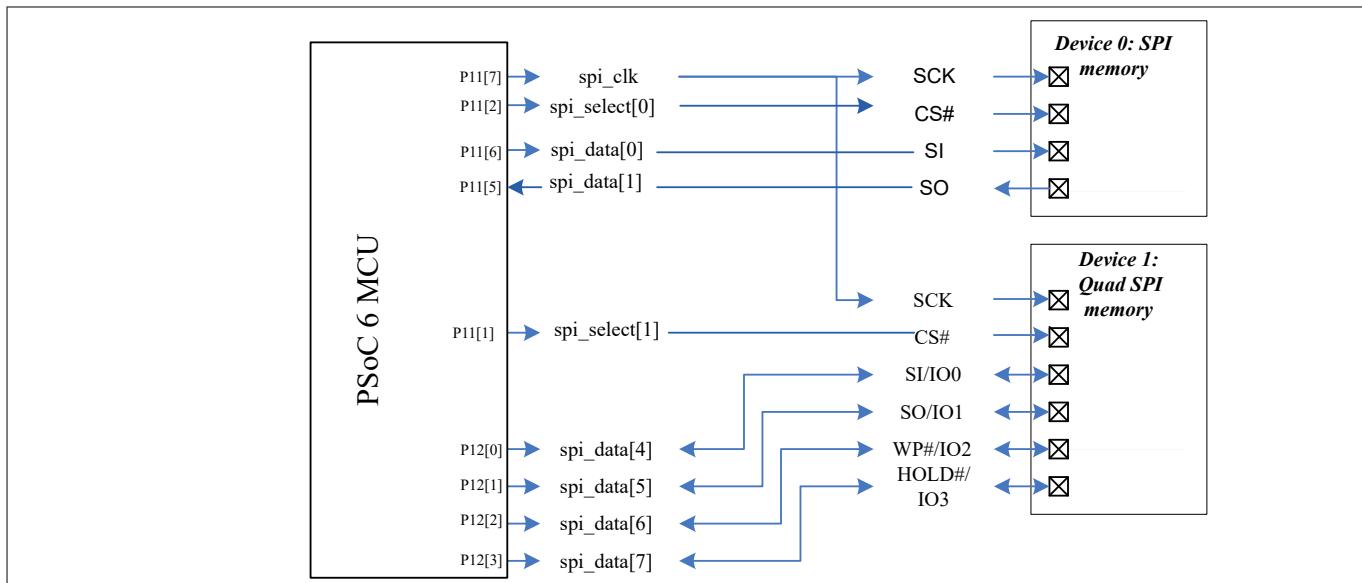


Figure 511 Serial memory interfacing with PSoC™ 6 MCU

~~5 PSoC™ 6 application notes~~

~~5.13.10~~ USB connection

The USB block is available as a fixed-function digital block in the PSoC™ device. It supports full-speed communication (12 Mbps) and is designed to be compliant with USB 2.0. The USB block includes the transmitter and receiver, which correspond to the USB physical layer (USB PHY). The USB PHY in PSoC™ also includes the pull-up resistor on the D+ line to identify the device as Full-Speed type to the host. The PHY integrates the 22- Ω series termination resistors on the USB lines.

5.13.10.1 PSoC™ 6 MCU USB pin description

Signal	PSoC™ 6 MCU pin	Functionality
USBDP (D+)	P14[0]	Data line
USBDM (D-)	P14[1]	Inverted data line
VBUS	VDDUSB	USB power supply
GND		Ground

5.13.10.2 PSOC™ 6 MCU as USB device

When designing hardware for a USB Device, consider the following:

- Use ferrite beads for VBUS, GND, and receptacle shield
- Use an ESD protection device placed near the USB receptacle

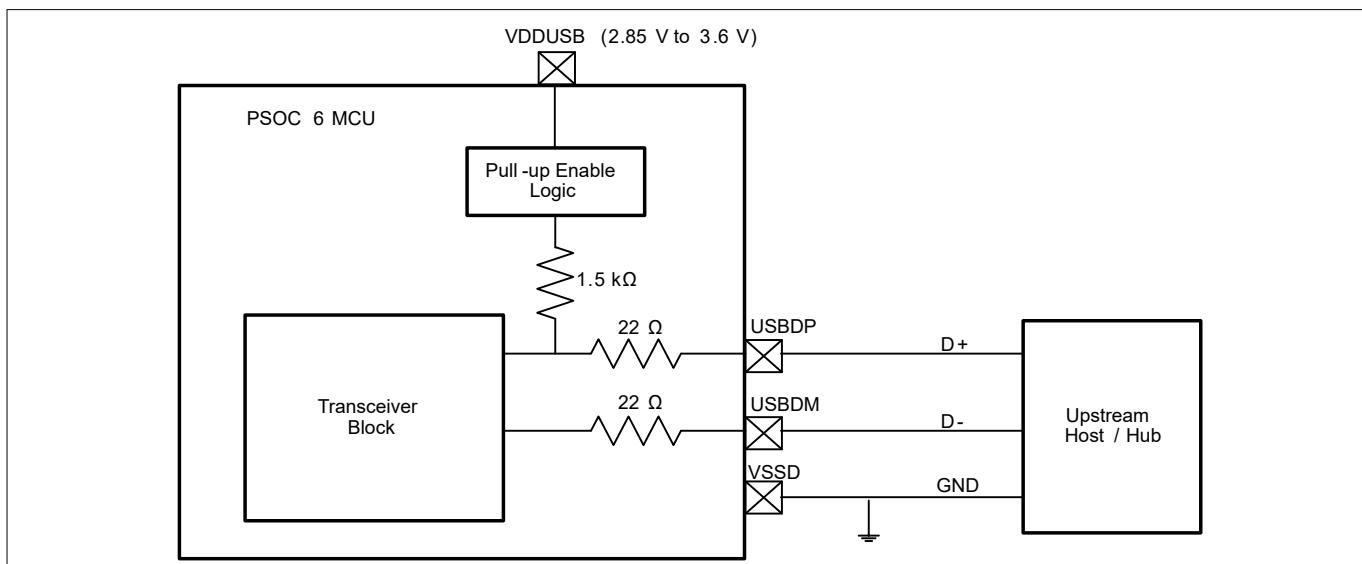


Figure 512 PSoC™ as USB device

VDDUSB powers the USBDM and USBDP pins. For proper operation of the USB block, the VDDUSB supply voltage must be in the range of 2.85 V to 3.6 V. When the USB block is not used in the design, the USB D+ and D- pins (USBDM and USBDP pins) can be used as GPIOs. When used as GPIOs, VDDUSB supports an operating voltage of 1.7 V to 3.6 V.

5 PSoC™ 6 application notes

~~DRWAF~~ 5.13.11 Antenna design

PSoC™ 63 family of devices includes an on-chip BLE radio. Therefore, antenna design and RF layout become critical for a wireless system that transmits and receives electromagnetic radiation in free space. The wireless range that an end-customer gets out of the system with a current-limited power source such as a coin-cell battery depends greatly on the antenna design, the enclosure, and a good PCB layout. [Table 107](#) provides quick guidelines on antenna and RF layout.

Table 107 Antenna and RF layout quick guide

PCB Stack up	Four-Layer PCB Infineon strongly recommends 4-layer PCB for all RF designs. Top Layer: RF IC and components, RF Trace, antenna, decoupling capacitors Layer 2: Ground plane Layer 3: Power plane Bottom Layer: Non-RF components and signals	Two-Layer PCB Typically used for simpler and cost-effective applications Top Layer: RF IC and components, RF Trace, antenna, decoupling capacitors Bottom Layer: Solid ground plane
Antenna Placement	Always place the antenna in a corner of the PCB with sufficient clearance as mentioned in the antenna datasheet from the rest of the circuit. Always follow the antenna designer's/manufacturer's recommended ground pattern for the antenna. Never place any component, planes, mounting screws, or traces in the antenna keep-out area across all layers. Ensure that the battery cable or mic cable does not cross the antenna trace on the PCB on the either side of the antenna. Do not place the antenna close to plastic in the industrial design. Proximity of plastic reduces the resonant frequency. Ensure that there is no trace or metallic plane in the antenna keep out area in any of the layers. Antenna must not be covered by a metallic enclosure Ensure that the orientation of the antenna is in line with the final product orientation so that the radiation is maximized in the desired direction. Plan to have a provision for an antenna-matching network because a lot of parameters in the antenna's proximity can vary its impedance, and therefore, the antenna may need retuning. Verify the antenna matching with the final enclosure	

(table continues...)

DRAFT

5 PSoC™ 6 application notes

Table 107 (continued) Antenna and RF layout quick guide

RF Trace Layout	<p>Choose the right kind of transmission line (micro strip or Coplanar waveguide (CPWG)) when calculating the trace width needed for a 50-ohm characteristic impedance.</p> <p>Ensure that the RF trace has a 50-Ω characteristic impedance and maintain constant width for the RF trace.</p> <p>Ensure a clean, uninterrupted ground beneath the RF trace without any other traces crossing the RF trace.</p> <p>Maintain the shortest possible length for the RF trace.</p> <p>Avoid bends in the RF trace. If bends are unavoidable, make a curved bend to maintain a uniform width.</p> <p>Avoid stubs or branching and test points on the RF trace.</p> <p>Do not place any other traces close to and parallel to the RF trace</p>
Ground Plane	<p>Allow a wide ground plane beneath the RF trace. It is better to keep a layer completely dedicated for ground.</p> <p>Keep the bottom ground plane together with the top ground plane and add vias between the two ground planes to improve the EMI and EMC performance.</p> <p>Cover the corners of the power plane with via holes connecting ground planes on either side of the power plane to arrest the unwanted EMI</p>
Power Supply Decoupling	<p>Place the capacitors close to the supply pin on the same layer as the IC with the smallest value capacitor closest.</p> <p>Use separate vias to the ground for each decoupling capacitor</p>
Vias	<p>Use plenty of vias spaced not more than one-twentieth of the wavelength of the RF signals between ground fillings at the top layer and inner ground layer.</p> <p>Place ground vias immediately next to pins/pads in the top layer and never share via with multiple pins/pads.</p> <p>Avoid using vias to route the RF trace to a different layer.</p> <p>Whenever possible, use vias to form a ground fencing around the RF section to isolate it from the rest of the circuit</p>
Capacitors and Inductors	<p>Use only C0G/NP0 capacitors for matching network and crystal load. X5R or X7R capacitors can be used for decoupling capacitors.</p> <p>Use only high-Q capacitors for RF circuits.</p> <p>For matching networks, use high-Q inductors with a self-resonant frequency (SRF) well above the operating frequency.</p> <p>For power supply filtering, use inductors with an SRF close to the noise frequency.</p> <p>Do not place inductors parallel and close to each other</p>
Coexistence with Wi-Fi	<p>Keep the BLE antenna and Wi-Fi antenna as far apart as possible.</p> <p>For antennas with linear polarization, orient the antennas such that they are electrically orthogonal to each other.</p> <p>If possible, orient the antennas such that the direction of the nulls of the antennas is collinear.</p> <p>Place via fencing between the BLE and Wi-Fi sections of the board to minimize leakage through the PCB</p>

5 PSoC™ 6 application notes

5.13.11.1 Support for external power amplifier/low-noise amplifier/RF front-end

~~DRAFT~~

Some applications may need a range higher than what is typically supported by the chipset. In such cases, either an external power amplifier (PA) and/or a low-noise amplifier (LNA) can be used to boost the link budget. For the 2.4-GHz radio, there are plenty of front-end ICs that include the power amplifier, low-noise amplifier, switches and controls needed to control them. These controls need to be precisely timed based on the actual transmit and receive timing. If the product has to remain BLE-compliant, ensure that the transmit power level does not exceed 20 dBm.

PSoC™ 6 MCU has three control signals to control the RF front-end ICs. [Figure 513](#) shows the interfacing of these control signals between PSoC™ 6 MCU and external RF front-end IC.

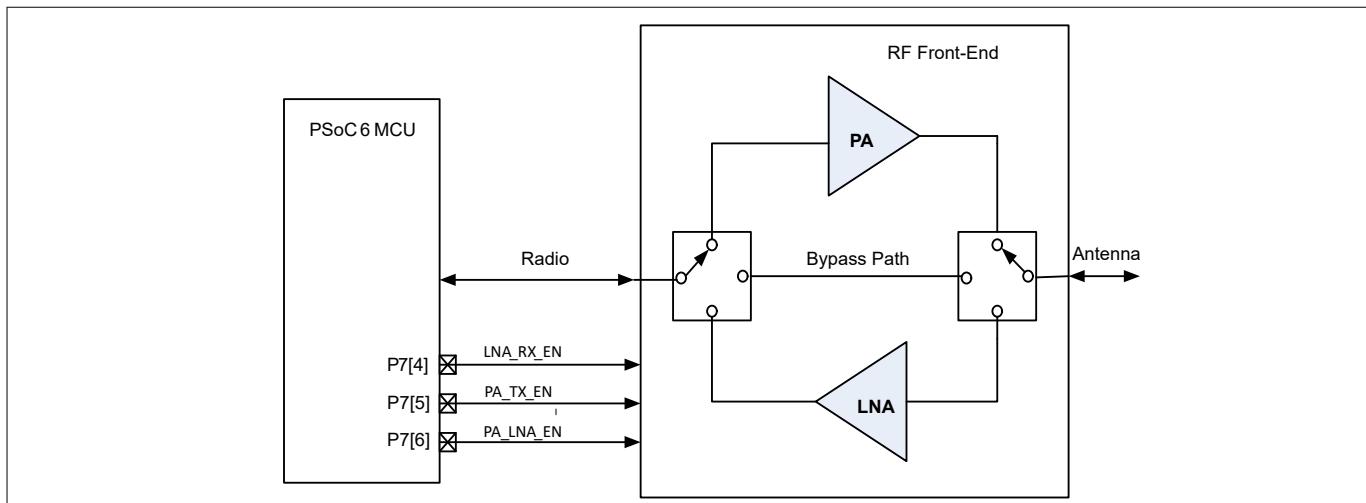


Figure 513 **Interfacing RF front-end with PSoC™ 6 MCU**

[Table 108](#) lists the control signals, their functions, and the corresponding pin mapping.

Table 108 **PSoC™ 6 MCU control signals for interfacing RF front-end IC**

Control signal	Port#[pin#]	Function
LNA_RX_EN	P7[4]	This signal is asserted when the receiver is enabled and de-asserted when receiver is off
PA_TX_EN	P7[5]	This signal is asserted when the transmitter is enabled and de-asserted when transmitter is off
PA_LNA_EN	P7[6]	This signal is asserted when either transmitter or receiver is active and de-asserted when neither of them is active. Can be used as chip select for front end modules

5 PSoC™ 6 application notes

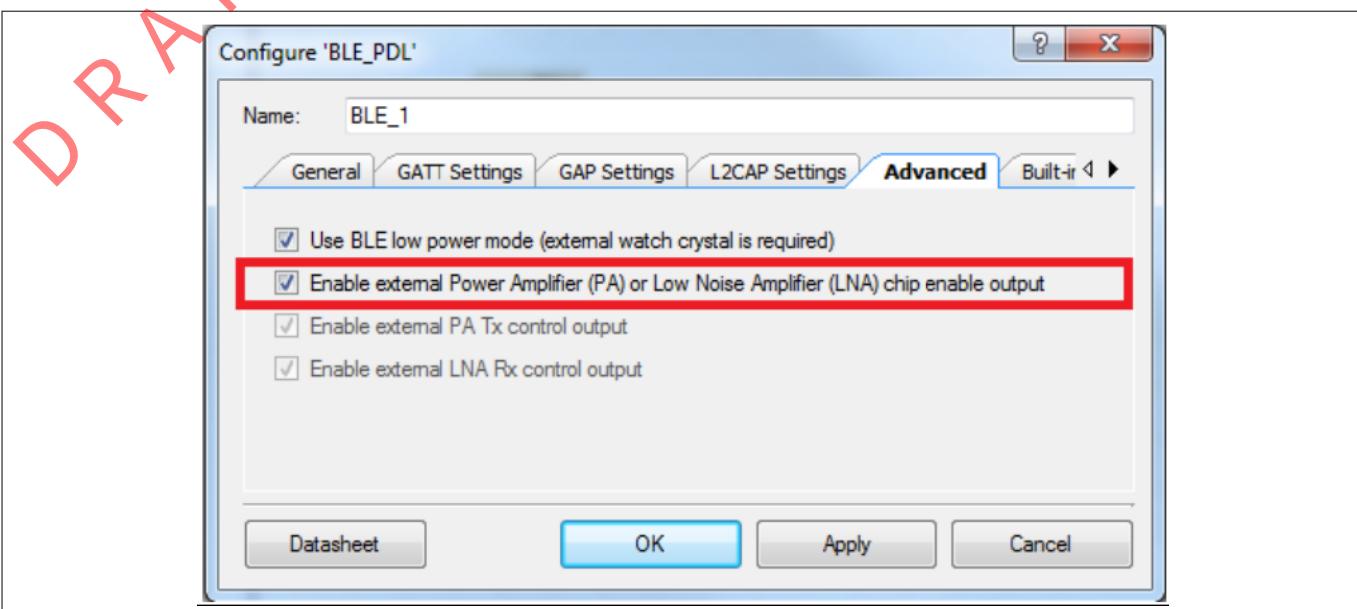


Figure 514 Configuring BLE to interface with external RF front-end IC in PSoC™ Creator

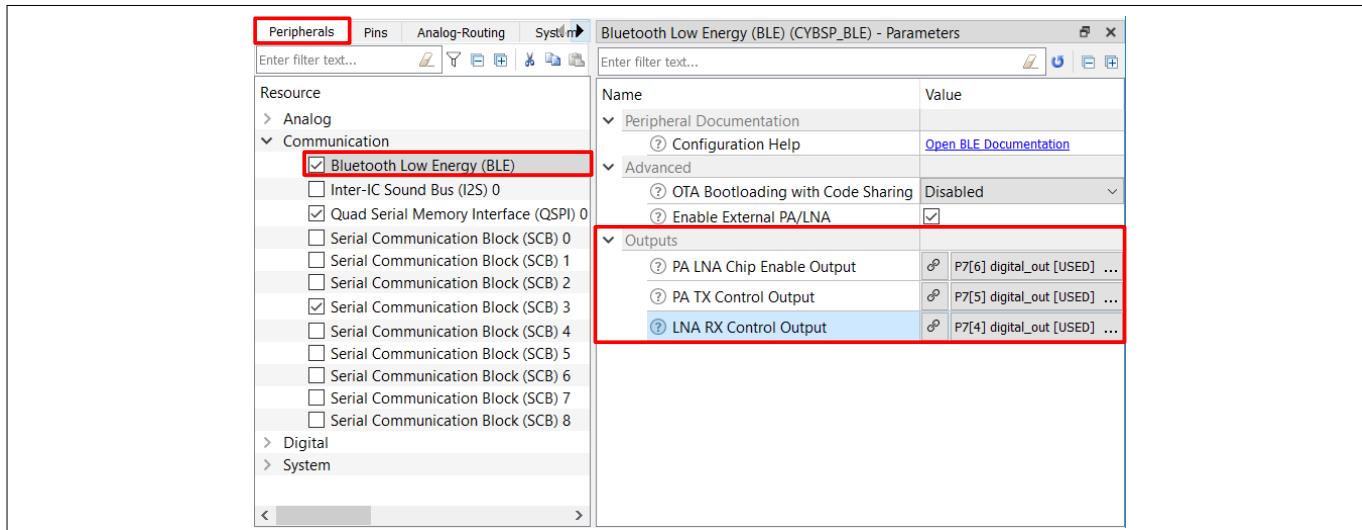


Figure 515 Configuring BLE to interface with external RF front-end IC in ModusToolbox™

You can configure the BLE Component to interface with external RF front-end as shown in [Figure 514](#) and [Figure 515](#) using PSoC™ Creator and ModusToolbox™ IDE respectively. For more details on antenna design, see [AN91445 – Antenna Design and RF Layout guidelines](#).

~~5 PSoC™ 6 application notes~~

~~5.13.12~~ **Audio subsystem**

The audio subsystem in PSoC™ 6 MCU consists of up to two I2S block and two PDM channels. The PDM channels interface to a PDM microphone's bit-stream output and produce word lengths of 16 to 24 bits at audio sample rate of up to 48 ksps. The I2S interface supports Master mode with Word Clock rates of up to 192 ksps (8-bit to 32-bit words).

Figure 516 and Figure 517 show the interfacing of PDM audio device and I2S audio device with PSoC™ 6 MCU, respectively.

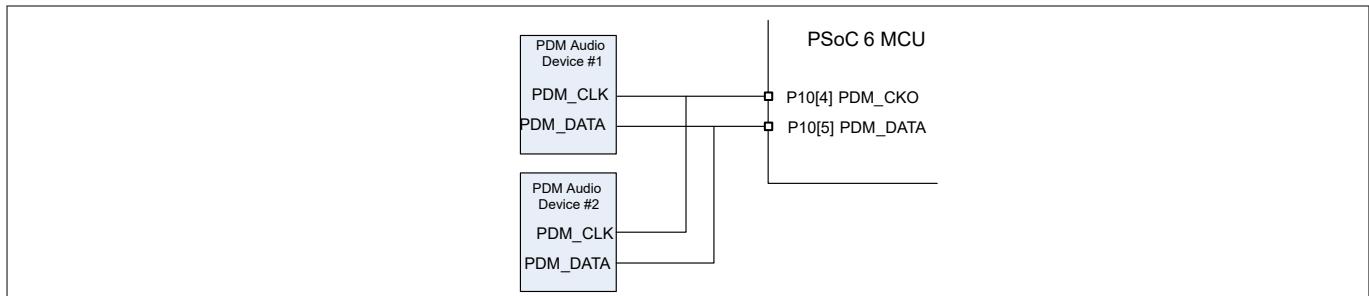


Figure 516 **Interfacing PDM audio device with PSoC™ 6 MCU**

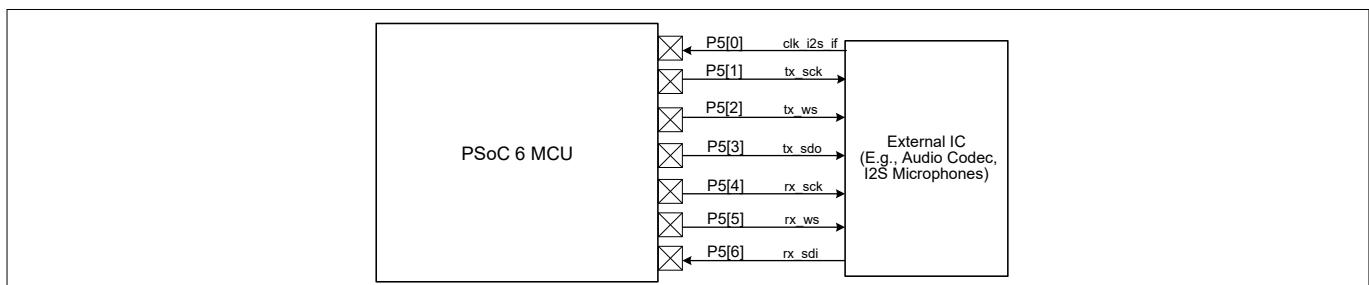


Figure 517 **Interfacing I2S audio device with PSoC™ 6 MCU**

I2S can be operated from an external master clock provided through an external IC such as audio codec. Pin P5[0] (clk_i2s_if) is used to drive an external clock to the I2S block.

5.13.12.1 **Clock generation for PDM-PCM converter**

In PSoC™ 6 MCU PDM-PCM converter has three stages of clock dividers to generate the clock (PDM_CKO), which is the input to the external PDM microphone clock input. The three stages are as follows:

- The first stage clock divider is used to generate the actual clock signal (PDM_CLK) which goes to the PDM-PCM converter. HFClk1 is the input clock for this stage. 1st Clock Divisor value can take integer values from 1 to 4
PDM_CLK=HFClk1/1st Clock Divisor
- The second stage is used to generate an internal master clock (MCLK) from the PDM_CLK. 2nd Clock Divisor value can take integer values from 1 to 4
MCLK=PDM_CLK/2nd Clock Divisor
- The third stage clock divider is used to generate the clock that goes to the PDM microphone. The third stage divider can take value between 2 and 16
PDM_CKO=MCLK/3rd Clock Divisor

The sample rate (F_s) for the PDM audio devices is given by the following relation:

$$F_s = \text{PDM_CKO} / (2 \times \text{Sinc Decimation Rate})$$

5 PSoC™ 6 application notes

~~DRAFT~~ 5.13.12.2 Clock generation for I2S audio devices

Audio applications require high accuracy clocks and therefore, a highly accurate ECO is required in such applications. Typically, a 17.2032-MHz crystal oscillator is used to generate 22.579 MHz for the 44.1-kHz audio sample rate and 24.576 MHz for the 48-kHz audio sample rate. [Table 109](#) shows the settings of the PLL to generate the required clock frequencies. You can set the divider and multiplier settings of the PLL either manually or automatically. [Table 110](#) shows the clock divider settings for typical audio sample rate and word lengths.

Table 109 PLL multiplier and divider settings

ECO (MHz)	PLL multiplier (P)	PLL divider (Q)	PLL output frequency (MHz)
17.2032	21	16	22.579
17.2032	10	7	24.576

Table 110 Clock divider settings for typical audio sample rates and word lengths

Audio sample rate (kHz)	PLL output frequency (MHz)	Word length					
		8-bit		16-bit		32-bit	
		Codec clock (kHz)	Clock divider	Codec clock (kHz)	Clock divider	Codec clock (kHz)	Clock divider
44.1	22.579	705.6	32	1411.2	16	2882.4	8
48	24.576	768	32	1536	16	3072	8
96	24.576	1536	16	3072	8	6144	4
192	24.576	3072	8	6144	4	12288	2

For more details, see the "PDM-PCM" and "I2S Sound Bus" chapters of [PSoC™ 6 MCU: Architecture TRM](#).

~~5 PSoC™ 6 application notes~~

~~5.13.13~~ Secure digital host controller

CY8C62XA/8/5 devices of the PSoC™ 6 MCU family have up to two secure digital host controllers (SDHC). The SDHC allows interfacing with embedded multimedia card (eMMC)-based memory devices, secure digital (SD) cards and secure digital input output (SDIO) cards. The SDHC block can be used to connect devices providing the SDIO interface, such as Infineon Wi-Fi products. [Figure 518](#) and [Figure 519](#) shows interfacing of a Wi-Fi device and SD storage with PSoC™ 6 MCU. It is recommended to have pull-up resistors (R_{pull_up}) in the range of $10\text{ k}\Omega - 100\text{ k}\Omega$ on the SDIO lines. Also, it is recommended to use series termination resistor of $33\text{ }\Omega$ on the SDIO lines.

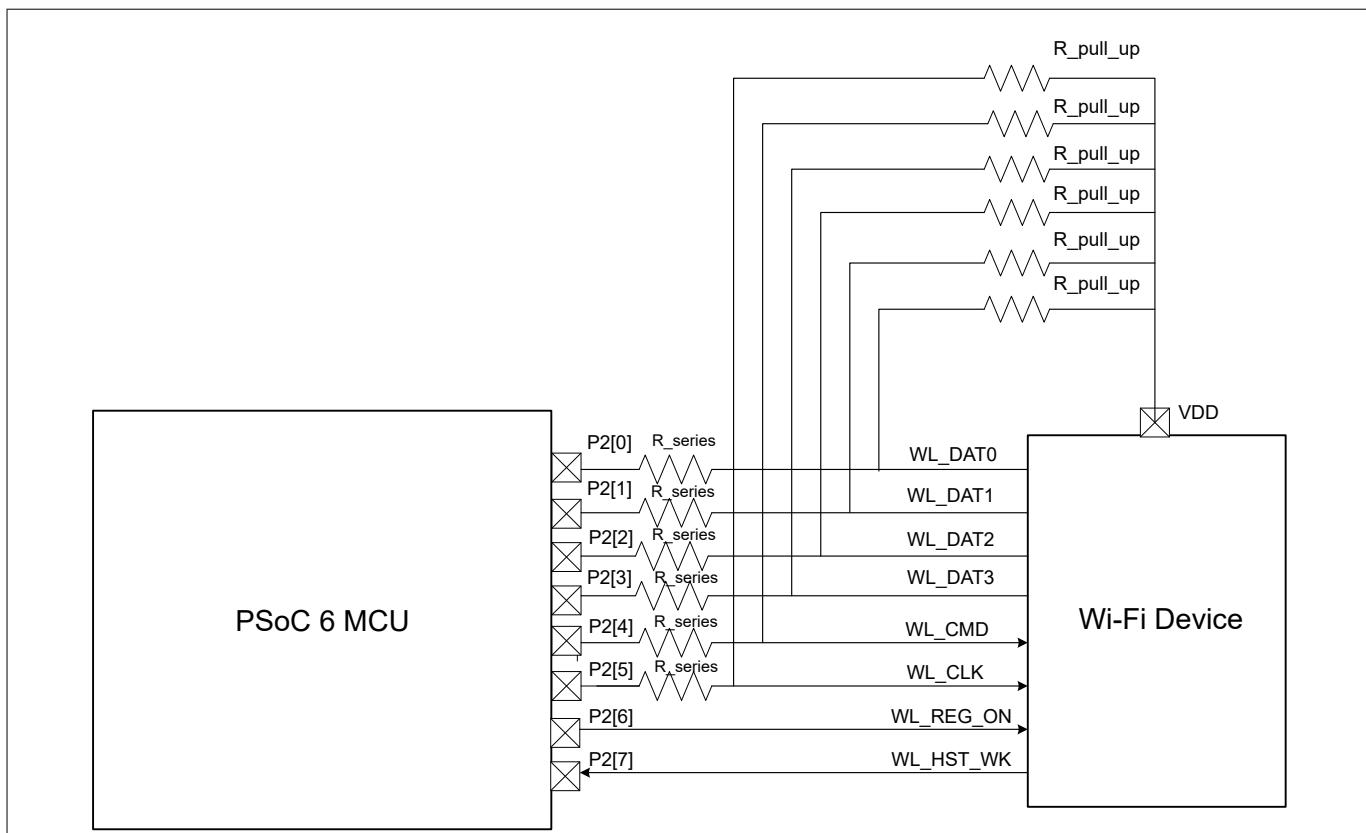


Figure 518 **Interfacing Wi-fi device using SDHC in PSoC™ 6 MCU**

5 PSoC™ 6 application notes

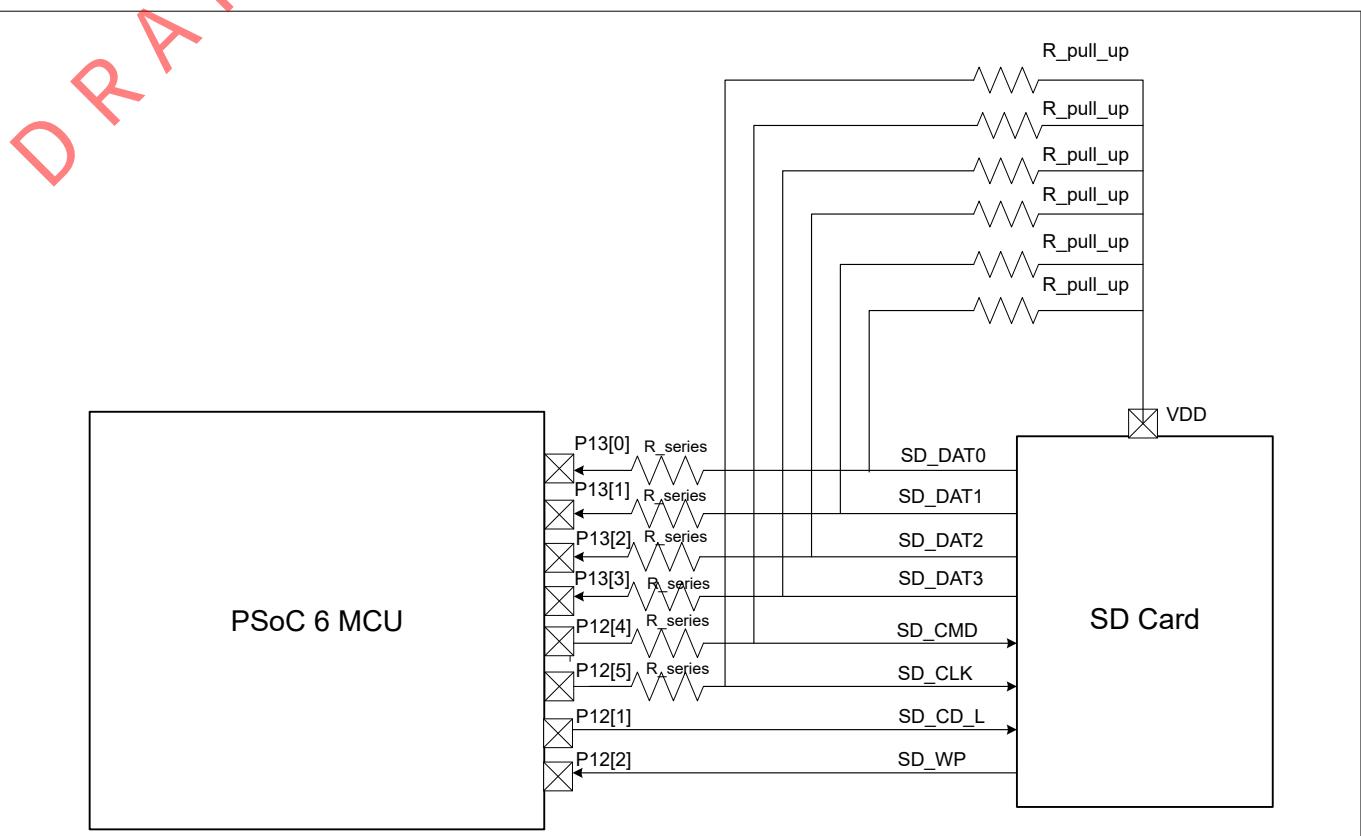


Figure 519 Interfacing SD storage using SDHC in PSoC™ 6 MCU

5 PSoC™ 6 application notes~~DRAFT~~
5.13.14 Summary

PSoC™ 6 MCU provides a flexible solution for designing digital and analog applications. This application note documented the considerations that you need to keep in mind when you build a hardware system around PSoC™ 6 MCU.

5 PSoC™ 6 application notes**References****Table 111 Application notes**

Document	Document name
AN79938	Design Guidelines for Infineon BGA Packaged Devices
AN69061	Design, Manufacturing, and Handling Guidelines for Infineon Wafer Level Chip Scale Packages
AN228571	Getting Started with PSoC™ 6 MCU on ModusToolbox™
AN221774	Getting Started with PSoC™ 6 MCU on PSoC™ Creator
AN210781	Getting Started with PSoC™ 6 MCU with Bluetooth® Low Energy (BLE) Connectivity
AN80994	Design Considerations for Electrical Fast Transient (EFT) Immunity
AN57821	PSoC™ 3, PSoC™ 4, and PSoC™ 5LP Mixed-Signal Circuit Board Layout Considerations
AN91445	Antenna Design and RF Layout Guidelines
AN91184	PSoC™ 4 BLE – Designing BLE Applications
AN95089	PSoC™ 4/PRoC BLE Crystal Oscillator Selection and Tuning Techniques
AN85951	PSoC™ 4 and PSoC™ Analog Coprocessor CAPSENSE™ Design Guide

5 PSoC™ 6 application notes

A PCB layout tips

Before beginning a PCB layout for PSoC™, it is a good idea to look at [AN57821 – PSoC™ Mixed - Signal Circuit Board Layout Considerations](#). Appendix A of that application note shows example PCB layouts and schematics for various PSoC™ packages.

There are many classic techniques for designing PCBs for low noise and EMC. Some of these techniques include:

- **Multiple layers:** Although they are more expensive, it is best to use a multilayer PCB with separate layers dedicated to the V_{SS} and V_{DD} supplies. This gives good decoupling and shielding effects. Separate fills on these layers should be provided for V_{SSA}, V_{SSD}, V_{DDA}, V_{DDIO}, and V_{DDD}.
To reduce cost, a two-layer or even a single-layer PCB can be used. In that case, you must have a good layout for all V_{SS} and V_{DD}.
- **Ground and power supply:** There should be a single point for gathering all ground returns. Avoid ground loops, or minimize their surface area. All component-free surfaces of the PCB should be filled with additional grounding to create a shield, especially when using two-layer or single-layer PCBs.
The power supply should be close to the ground line to minimize the area of the supply loop. The supply loop can act as an antenna and can be a major emitter or receiver of EMI.
- **Decoupling:** The standard decoupler for external power is a 100- μ F capacitor. Supplementary 0.1- μ F capacitors should be placed as close as possible to the V_{SS} and V_{DD} pins of the device to reduce high-frequency power supply ripple.
Generally, you should decouple all sensitive or noisy signals to improve the EMC performance. Decoupling can be both capacitive and inductive.
- **Component position:** Separate the circuits on the PCB according to their EMI contribution. This will help reduce cross-coupling on the PCB. For example, separate noisy high-current circuits, low-voltage circuits, and digital components. The decoupling capacitors and the inductor (Buck Inductor) should be placed as close as possible to the device pins with minimum trace resistance.
- **Signal routing:** When designing an application, the following areas should be closely studied to improve the EMC performance:
 - Noisy signals. For example, signals with fast rise times
 - Sensitive and high-impedance signals
 - Signals that capture events, such as interrupts and strobe signalsTo improve the EMC performance, keep the trace lengths as short as possible and isolate the traces with V_{SS} traces. To avoid crosstalk, do not route them near to or parallel to other noisy and sensitive traces.

For more information, several references are available:

- The Circuit Designer's Companion, Second Edition, (EDN Series for Design Engineers), by Tim Williams
- PCB Design for Real-World EMI Control (The Springer International Series in Engineering and Computer Science), by Bruce R. Archambeault and James Drewniak
- Printed Circuits Handbook (McGraw Hill Handbooks), by Clyde Coombs
- EMC and the Printed Circuit Board: Design, Theory, and Layout Made Simple, by Mark I. Montrose
- Signal Integrity Issues and Printed Circuit Board Design, by Douglas Brooks

5 PSoC™ 6 application notes
B ~~DATA SHEET~~ Schematic checklist

The answer to each that is in the following checklist should be Yes (Y) or Not Applicable (N.A.). For example, if you power a PSoC™ 6 MCU device with an unregulated external supply in your application, you can mark all the items of “Power (regulated external supply)” as N.A.

Catalog	Item	Y/N/N.A	Remark
Power	Are the power supply pin connections made in accordance with Power pin connections		
	Are the 0.1- μ F and 1- μ F capacitors connected to each VDDD, VDDIO, VDDA, or VDDR pins?		
	Are the 10- μ F and 0.1- μ F capacitors connected to VDD_NS pin?		
	Are the voltages (including ripples) at the VDDD and VDDA pins in the range of 1.7 to 3.6 V?		
	Is the VCCD pin connected to a 4.7 μ F capacitor?		
	If the VRF pin powers BLE, is the pin connected to a 10 μ F capacitor?		
	For CY8C61x6/7, CY8C62x6/7, CY8C63x6/7 devices, is the 2.2- μ H inductor connected between VIND1 and VIND2 pins?		
	For CY8C61xA/8, CY8C62xA/8, CY8C61x5, CY8C62x5, CY8C62x4 devices, is the 2.2- μ H inductor connected between VIND and VCCD pins?		
	Is the VRF pin connected to the VDCDC pin?		
	Is the VDDR pin connected to the VDCDC pin?		
Clocking	Is the VBACKUP pin connected to an appropriate supply (VDDD or the 1.4-V to 3.6-V source)?		
	Is the 1- μ F capacitor connected to the VDDR_HVL pin, and has no external load?		
	Is the external clock connected to EXT_CLK pin?		
	Is the external clock's frequency less than or equal to 48 MHz (including tolerance)?		
	Is the external clock's duty cycle from 45 percent to 55 percent?		
	Is the external 32-MHz crystal connected to XI and XO for BLE operation?		
	Is the external MHz crystal connected to ECO pins for ECO operation?		
Reset	Is the 32.768-kHz crystal connected to WCO for RTC operation? Are the WCO load capacitors connected?		
	Is the external 32.768-kHz square wave clock connected to WCO_OUT pin and WCO_IN left floating?		
Programming and debugging	Is the reset pin connection made in accordance with Figure 499 ?		
Programming and debugging	Are the SWD/JTAG/ETM signals connected as described in the Programming and debugging section?		

5 PSoC™ 6 application notes

DRAFT

Catalog	Item	Y/N/N.A	Remark
GPIO pins	<p>Is the assignment of your GPIO pins done in the sequence described in I/O pin selection?</p> <p><i>Note:</i> <i>Simultaneous GPIO switching with unrestricted drive strengths and frequency can induce noise in on-chip subsystems affecting CAPSENSE™ and ADC results. See to the Errata section in the respective device datasheet for details.</i></p>		
	Is every GPIO pin's sink current lower than 8 mA?		
	Is every GPIO pin's source current lower than 4 mA?		
	Is the GPIO pins' total source current or sink current smaller than device capability?		
SCB	Is the assignment of the SCB's fixed pins in accordance with the device datasheet ?		
CAPSENSE™	<p>Are the pins with strong sink current kept away from the CAPSENSE™ pins (the space is more than three pins)?</p> <p>Is C_{MOD} connected to the CMOD (or C_MOD) pin for self-capacitive sensing?</p> <p>Is C_{SH_TANK} connected to the CTANK (or C_SH_TANK) pin for self-capacitive sensing as explained in the CAPSENSE™ section?</p> <p>Are the C_{INT1} and C_{INT2} capacitors connected for mutual capacitive sensing?</p> <p>Are the CAPSENSE™ sensor, shield, Rx, and Tx signals selected based on preference provided in Table 103?</p>		
Antenna	Is the Antenna design based on recommendations from AN91445 ?		

~~5 PSoC™ 6 application notes~~

~~5.13.17 Revision history~~

Document version	Date of release	Description of changes
**	12/27/2016	New application note
*A	08/16/2017	Updated template. Updated Support for External Power Amplifier/Low-Noise Amplifier/RF Front-End. Updated Figure 483 Updated Table 100
*B	04/27/2018	Updated Figure 483 , Figure 485 , Section 3.2, and Appendix B
*C	11/01/2018	Updated for ModusToolbox™
*D	04/09/2019	Updated the following sections: GPIO pins , CAPSENSE™ , and SAR ADC Updated decoupling capacitor values in the Power pin connections section
*E	09/16/2019	Added CY8C62x5 device support; Updated for ModusToolbox™ 2.0
*F	12/16/2019	Updated WCO load capacitor connection diagram (Figure 492)
*G	03/27/2020	Updated ECO frequency range to 16 MHz to 35 MHz Updated ECO and WCO load capacitor calculation. Removed ECO configuration in ModusToolbox™ and PSoC™ Creator figures. Updated ECO load capacitor connection diagram (Figure 490) Updated WCO load capacitor connection diagram (Figure 492)
*H	07/23/2020	Added CY8C62x4 device. Updated Power pin connections – Added recommendations for buck regulator inductor and capacitor selection; Updated Figure 483 and Figure 484 . Updated SAR ADC – ADC sampling rate and acquisition time. Updated PILO accuracy
*I	2021-03-08	Updated to Infineon Template
*J	2022-07-21	Template update

5.14 AN219434 PSoC™ 6 MCU importing generated code into an IDE

About this document

-
- 1
- 4

Scope and purpose

AN219434 shows how to import the code generated by PSoC™ Creator, for the PSoC™ 6 MCU architecture, into your preferred integrated development environment. With this knowledge, you can combine the benefits of automatically generated code with your preferred IDE. High-level options are explained, with detailed instructions for each option.

Associated part family

PSoC™ 6 MCU

5 PSoC™ 6 application notes

Associated code examples

[CE212726](#)

Related application note

[AN210781](#)

More code examples? We heard you.

To access an ever-growing list of hundreds of PSoC™ code examples, please visit our [code examples web page](#). You can also explore the video training library [here](#).

~~5 PSoC™ 6 application notes~~

~~5.14.1~~ Introduction

A significant number of customers use the PSoC™ Creator tool as a hardware design platform but write application software in a different environment. There are several reasons why this might be. Your organization may have an established set of tools for developing products. You may wish to reuse legacy code built in another IDE. Or you may have a preferred development environment. This application note shows you how to use PSoC™ Creator generated code for the PSoC™ 6 MCU in such an IDE.

The PSoC™ 6 MCU is based on a dual-CPU Arm® Cortex®-M4 (CM4) and Cortex-M0+ (CM0+) Programmable System-on-Chip.

To enable firmware development for PSoC™ 6 MCU devices, Infineon provides two distinct development environments: one based on PSoC™ Creator, the other based on ModusToolbox™ v2.x software. [Table 112](#) summarizes the key features for each workflow. See [ModusToolbox™ Software Overview](#) of the [ModusToolbox™ User Guide](#) for a high-level look at what's available in ModusToolbox™ v2.x, and how it works.

Table 112 Comparing PSoC™ Creator and ModusToolbox™ software

Feature	PSoC™ Creator	Eclipse IDE for ModusToolbox™
IDE Framework	Proprietary, not extensible	Eclipse-based, extensible
Driver Library	Peripheral Driver Library (PDL)	psoc6pdl, psoc6hal library on GitHub
Code Generation	Components (including Bluetooth® Low Energy and CAPSENSE™)	Configurators: device, CAPSENSE™, Bluetooth®, QSPI, SmartIO, SegLCD, USBDev
Host OS	Windows	Windows, macOS, Linux
Middleware	Bluetooth® Low Energy, dfu, em_eeprom, emWin, usb_dev, FreeRTOS	Extensive collection of libraries for multiple ecosystems. See PSoC™ 6 Middleware on GitHub.
Post-processing tool	cymcuelftool (Windows, macOS, Linux)	Not required for most use cases; cymcuelftool is still supported

This application note discusses the PSoC™ Creator environment and workflow. [AN225588, Using ModusToolbox™ Software with a Third-Party IDE](#), discusses that environment and workflow.

The Peripheral Driver Library (PDL) is a software development kit (SDK) for PSoC™ 6 MCU devices. Your firmware uses PDL API function calls to configure, initialize, enable, and use a peripheral driver. The PDL also includes support for middleware such as Bluetooth® Low Energy, and real-time operating systems (RTOS). By design, the PDL is IDE-neutral.

However, PSoC™ Creator is fully integrated with the PDL. As a design environment, PSoC™ Creator helps you configure clocks, interrupts, pin assignments, and drivers using a friendly UI. To customize a driver, add a Component to the PSoC™ Creator design corresponding to the peripheral. Instead of writing configuration code, modify options in the Component. PSoC™ Creator then generates all the configuration code to set up the clocks, interrupts, pins, and custom drivers based on the design. When targeting a PSoC™ 6 MCU device, PSoC™ Creator generates code using the PDL. If you use the PDL without PSoC™ Creator, you must write all the configuration code for the design.

Generated code is a valuable resource because it handles a great deal of the complexity involved in setting up the firmware for a design. This application note explains how to use that generated code in any IDE. There is more than one way to do this. This note explores and explains your options. It also walks through an example to show you how it can be done. When you have finished this application note, you will know what your choices are, and how to integrate generated code into your preferred IDE.

Even if your circumstances prevent you from using generated code directly, all is not lost. This application note shows you how to use generated code to learn about the PDL.

To learn more about the PDL or PSoC™ Creator, see [References](#) for links to some of the available resources.

5 PSoC™ 6 application notes**5.14.2 Integrating generated code**

~~DRAFT~~ At a high level, there are two ways to integrate generated code into an IDE's project file. The two paths are:

- Export code from PSoC™ Creator and import that code into an IDE
- Manually add generated source files to an IDE's project

The flowchart in [Figure 520](#) shows the tasks you perform for each path. This application note explains the one-time tasks noted in the flowchart. Based on your circumstances, one or the other path may be better for you.

The export path is available only for supported IDEs. See [Exporting and importing generated code](#) for details about this path. Supported IDEs include:

- IAR Embedded Workbench
- Keil µVision
- Eclipse-based IDEs

Manually importing code works for any IDE, including supported IDEs. For supported IDEs, additional resources are available that make the manual approach easier, including a fully configured project file, IDE-specific startup code, flash configuration files, and linker scripts. See [Manually importing generated code](#) for details about this path.

5 PSoC™ 6 application notes

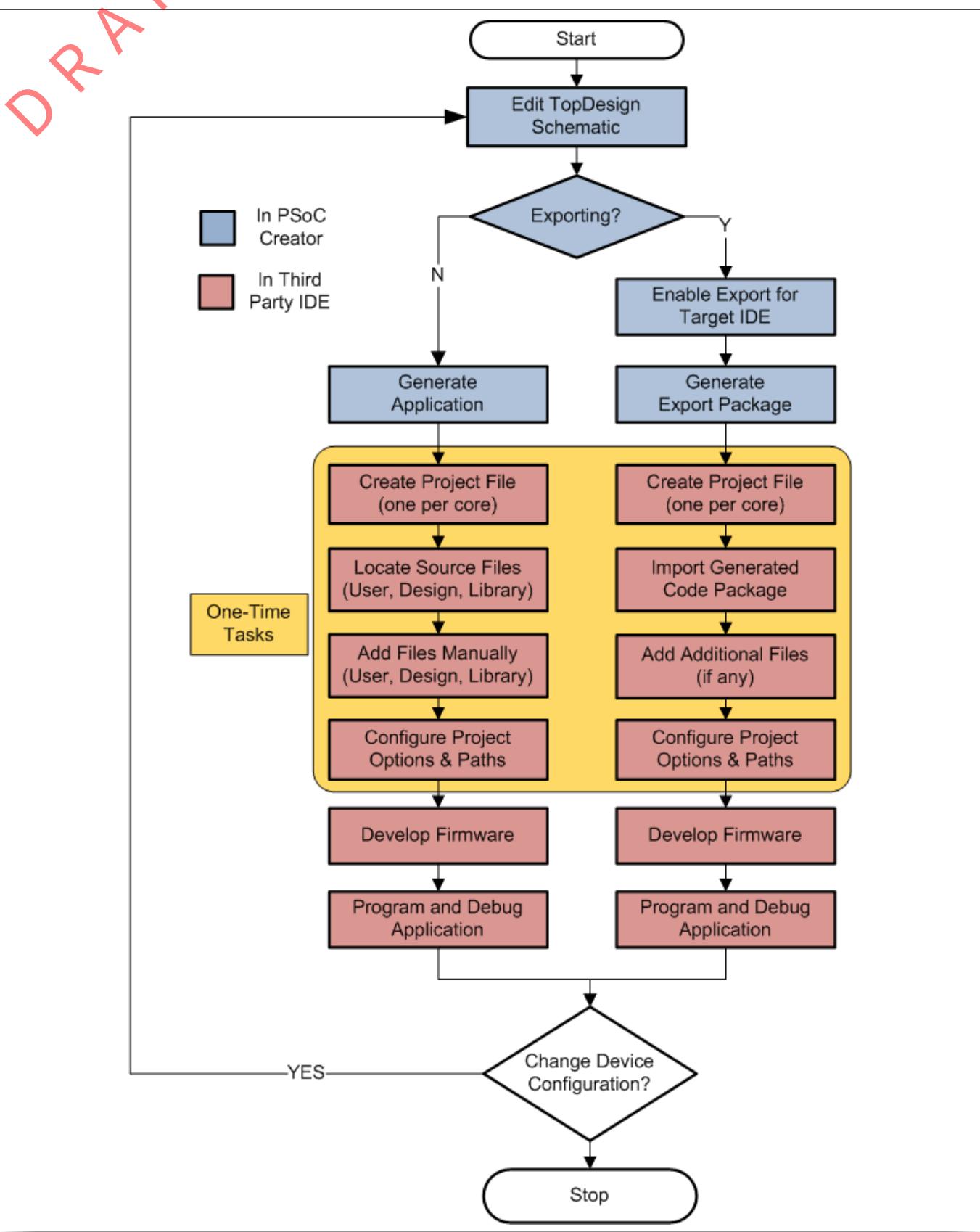


Figure 520

Integrating generated code

~~5 PSoC™ 6 application notes~~

~~5.14.2.1~~ Advantages of generated code

If you use PSoC™ Creator to design and configure your system, you gain significant benefits of generated code. All the necessary configuration code is created for you. This code can be quite complex. For example, the generated code in `cyfitter_cfg.c` initializes all clocks, sets each clock's source and divider, and enables all the clocks in your design. It makes PDL API function calls to do so. This code is based on the clock tree configuration in the PSoC™ Creator design.

The Bluetooth® Low Energy Component is another good example. The code to configure a Bluetooth® Low Energy Component is typically hundreds of lines long, declares a dozen configuration structures, and defines the values for more than 100 fields. PSoC™ Creator generates all this code automatically, based on the Bluetooth® Low Energy Component configuration in the design.

In addition to configuration code, PSoC™ Creator generates a simplified function API for each Component. The Component API is built on and uses the PDL API. This means that you can use the Component API instead of calling PDL functions directly. You can mix and match these APIs. They are consistent and compatible.

For example, to initialize, enable, and start a PWM using the PDL API, your code might look like this (ignoring errors):

```
Cy_TCPWM_PWM_Init (TCPWM1, counterNumber, &PWM_config);
Cy_TCPWM_Enable_Multiple (TCPWM1, whichCounters);
Cy_TCPWM_TriggerStart (TCPWM1, whichCounters);
```

By contrast, using the Component API, your code would look like this (for a TCPWM Component named MyPWM):

```
MyPWM_Start();
```

The code generator implements the proper sequence of function calls to enable the PWM. It also provides all the hardware-related parameters based on the design and configuration in PSoC™ Creator. If you examine the generated code for the `MyPWM_Start()` function, you find all three PDL API calls implemented for you, and called in the correct sequence.

The generated code is primarily in C source and header files but may include assembler files and compiled binaries. If you add these generated files to an IDE project correctly, as described in this application note, iterative development is fully supported. When you modify the design in PSoC™ Creator and regenerate the application, the IDE recognizes that files have changed with no additional work required.

5.14.2.2 Limitations of generated code

PSoC™ Creator generates code compatible with the C99 standard. The compiler you use must support that standard. In addition, compilers can vary in subtle implementation details. There is always a chance that a particular compiler may handle some syntax in a non-standard way, but this is unusual.

The biggest limitation is that in most cases you cannot modify generated code. If you modify the code in a generated source file, the change is lost when you regenerate the code. Changes you make to generated code do not migrate upstream into the PSoC™ Creator design. If during the development cycle you discover that you need to modify the design, make the changes in PSoC™ Creator, and regenerate the application.

However, certain files generated by PSoC™ Creator are treated as user files. You can modify any user file without losing changes. See [User files](#).

5 PSoC™ 6 application notes~~DRAFT~~
5.14.2.3 Tool compatibility

PSoC™ Creator and PSoC™ Programmer are proprietary tools built for Windows OS. When working with tools from other providers, you may encounter compatibility issues. For example, PSoC™ Creator adds proprietary information to the final hex file, and PSoC™ Programmer requires this information; other IDEs do not generate this information, so PSoC™ Programmer cannot use such a hex file.

ModusToolbox™ software and the newer Infineon Programmer do not have these issues.

5 PSoC™ 6 application notes

5.14.3 Exporting and importing generated code

The export/import path is available for supported IDEs only. For supported IDEs, PSoC™ Creator provides an export package. [Table 113](#) lists the export packages and which IDEs each package supports.

Table 113 Available export packages

Package	Supported IDE	Principal file	Path
CMSIS Pack	Keil µVision v5 or later IAR Embedded Workbench v8.1 or later Eclipse-based IDEs that can import a CMSIS Pack	.pack file	Export/Pack
Inter-Project Connection File (IPCF)	IAR Embedded Workbench	.ipcf file (one per CPU)	Export
Makefile	GNU command-line tools	makefile	<project>.cydsn

The export/import process is the same for all supported IDEs. After developing your design, take these steps:

- In PSoC™ Creator:
 - Enable the creation of the export package
 - Build your code (this generates the export package)
- In the 3rd party IDE:
 - Create an empty project (one per CPU for multi-CPU devices)
 - Import the package into the empty project
 - If necessary, add additional files not included in the export package
 - Configure project options

The details of the export/import process vary significantly among supported IDEs. This section gives you an understanding of how the process works. The exact steps and precise details are described in the PSoC™ Creator Help and [PSoC™ Creator User Guide](#). This application note does not duplicate this information. You must use the documentation to successfully export and import a package into an IDE.

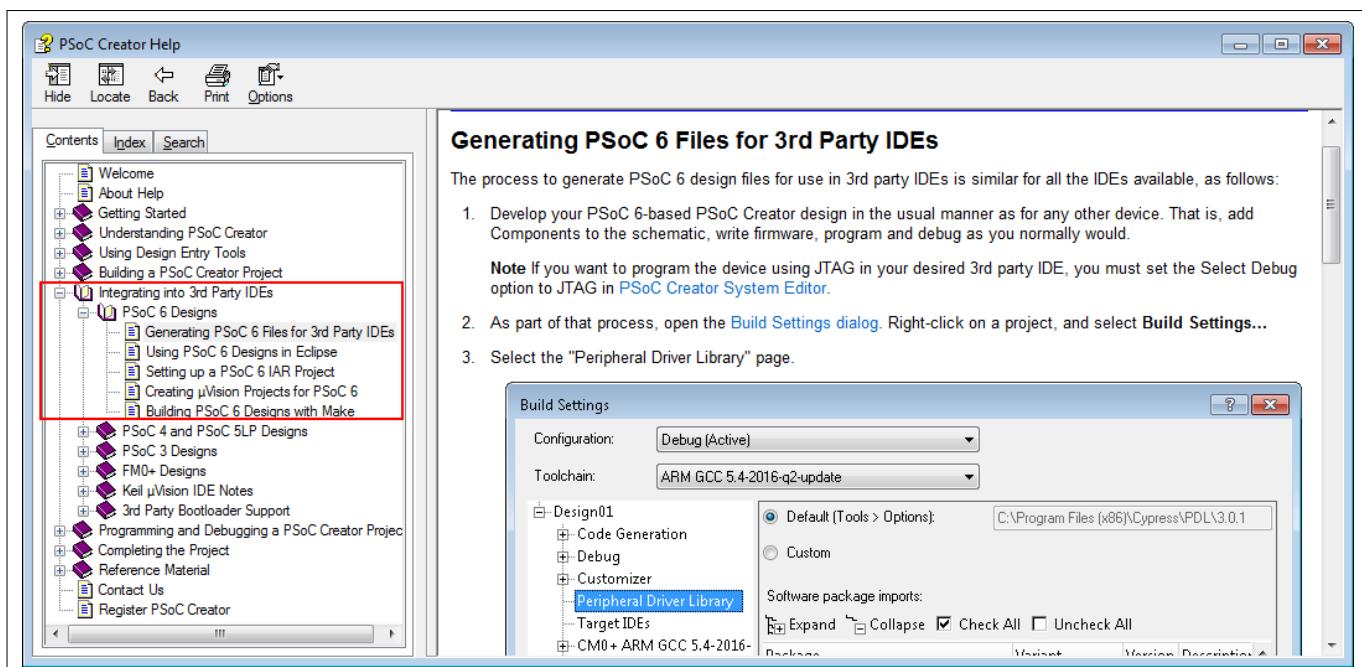


Figure 521 PSoC™ Creator help topics for integrating into third-party IDEs

~~5 PSoC™ 6 application notes~~

~~5.14.3.1~~ Enabling export for a target IDE

In PSoC™ Creator, use the **Project > Build Settings** menu command to open the project settings. Then, open the **Target IDEs** panel. Choose **Generate** for the export package you want to create. [Figure 522](#) shows the CMSIS Pack option enabled. A **CMSIS Pack** is an IDE-agnostic delivery package for software components.

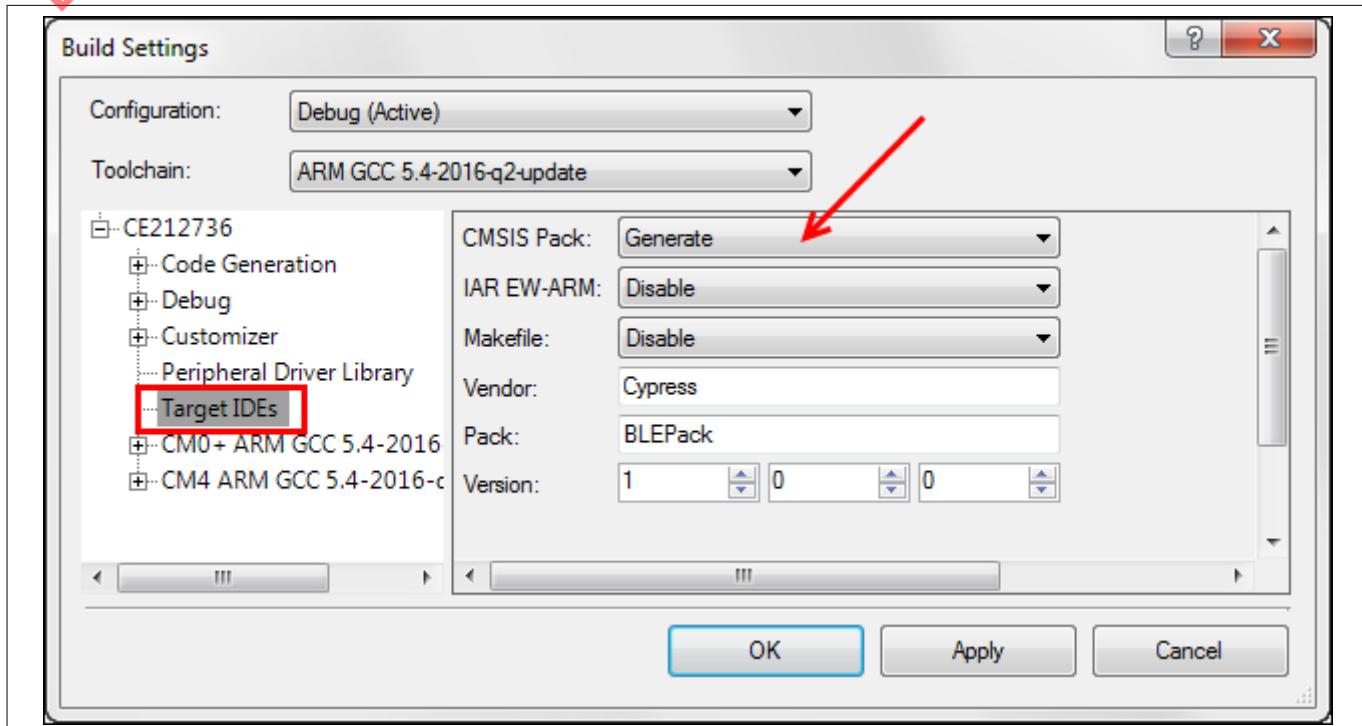


Figure 522 Enabling the CMSIS pack export package

As an example of how the export process varies, the **Vendor**, **Pack**, and **Version** information shown in the figure are required only for the CMSIS Pack. IAR EW-ARM has different options.

See PSoC™ Creator documentation for details.

5.14.3.2 Generating the export package

After enabling the creation of a package, build your code. Use the PSoC™ Creator **Build > Build <project>** command. Note that the **Build > Generate Application** command does not generate the export package.

The contents of the Export folder vary per package. The package is a collection of files. PSoC™ Creator puts each package at the location listed in [Table 113](#). When importing the package, navigate to that location to find the required files.

5.14.3.3 Importing the package

The import step is highly variable, based on the IDE and package. This section introduces you to the general tasks that you must perform to import a package. Refer to the PSoC™ Creator documentation for the precise steps and details. There is a help topic for each supported IDE, as shown in [Figure 521](#).

In general, you perform these tasks:

- Create an empty project file (one per CPU for multi-CPU devices)
- Import generated code into each project file. [Figure 523](#) shows where each package is located
 - For a CMSIS Pack, install the Pack on your computer, and then import the Pack
 - For IAR tools, establish a project connection using the .ipcf file

For GNU tools, use the generated makefile.

5 PSoC™ 6 application notes

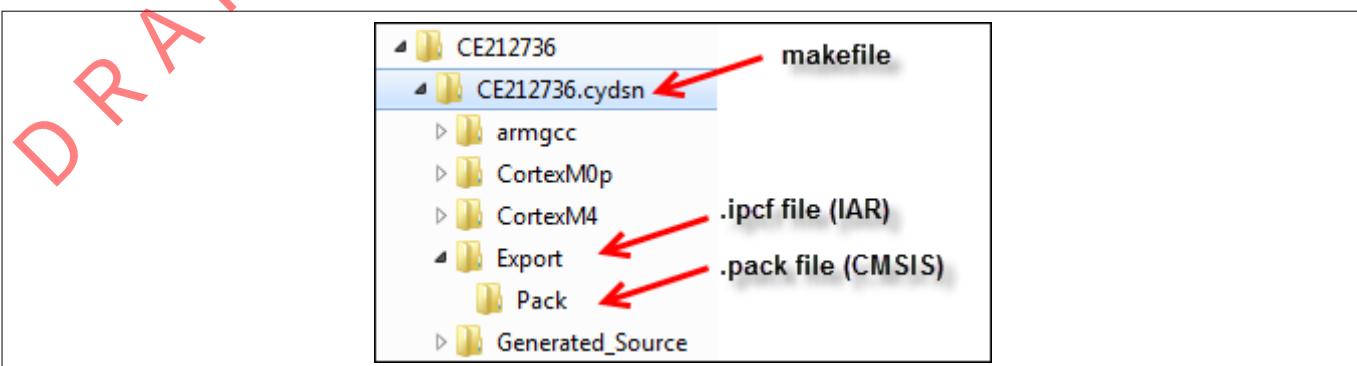


Figure 523 Where to find export packages

- In some cases, add additional files to each project file:
 - For CMSIS Pack, you can create the CPU-specific `main.c` and `linker` files from templates
 - For IAR tools, these files are included in the project automatically
- Set project options, which may include compiler, linker, and debugger settings

Different packages require that you set different project options. For example, the IPCF connection for IAR tools requires that you modify output folder names for the CM4 project, so they do not conflict with the CM0+ project. For the CMSIS Pack, you also specify include paths for header files, provide a path to the linker command file, and so on. PSoC™ Creator documentation provides all the details. Ensure that you are familiar with that documentation before exporting and importing generated code. See [Appendix A](#) for more information about project options.

Several files in a project are CPU- or IDE-specific, such as linker files, startup code, the makefile, and others. The correct files for each CPU and IDE are in the package. When you import the package, the correct files are included.

Note: *Many CPU-specific files are user-editable files, initially generated by PSoC™ Creator. You can modify these files and changes will not be lost when you rebuild the PSoC™ Creator project. See [User files](#) for more information.*

5.14.3.4 Supporting iterative development

After importing the generated code, you will likely encounter the need to change your design. Make the required change in the PSoC™ Creator project and rebuild the project. Do not modify files in the `Generated_Source` folder. Any change will be lost when you regenerate code.

When you rebuild the code in PSoC™ Creator, an updated package appears in the `Export` folder. The code generation process may have modified files, added new files, or removed files.

For the CMSIS Pack export, reinstall the Pack. Locate the pack file in the `Export` folder and double-click to install. The unzip dialog includes an alert that the Pack is already installed. Replace it. The IDE automatically recognizes the changes, including added or removed files. Rebuild your code in the IDE to update the executable.

The IAR inter-project connection file points to the PSoC™ Creator `Generated_Source` folder. The IDE automatically recognizes the changes, including added or removed files. There is no need to reconnect to the `.ipcf` file. Simply rebuild your code in the IDE to update the executable.

For a makefile, ensure that your build process uses the files in the `Export` folder. If it does, then a simple clean and build updates the executable using the new code.

5 PSoC™ 6 application notes~~DO NOT USE~~
5.14.3.5 Export/Import review

PSoC™ Creator documentation describes all the details. The import process begins with a new, empty project in the IDE.

For a CMSIS Pack, project setup for the IDE involves multiple steps. The Pack may be imported into a variety of IDEs, and each IDE is unique in how it handles build options, so you are required to set several options. This is a one-time task. Any change in the code (from a design change in PSoC™ Creator) is handled transparently. Just reinstall the Pack.

Inter-project connection is a protocol proprietary to the IAR tools. Because it is designed for one IDE, the project connection sets most options automatically. This is a very friendly process.

~~DO NOT USE~~

5 PSoC™ 6 application notes

5.14.4 Manually importing generated code

Manual import works for any IDE, including supported IDEs that have an export/import process. For a non-supported IDE, manual import is the only option.

To begin, you must have generated code. In PSoC™ Creator, the **Build > Generate** Application command is sufficient. You do not need to compile the code. To manually import generated code, you:

- Create a project in the IDE (one for each CPU for a multi-CPU device)
- Locate and identify the generated files you need
- Add those files to the project

This section gives you the background required to understand what files PSoC™ Creator generates, where to find them, and which you need to add to your project. [Manually importing generated code – an example](#) walks you through the process in detail.

5.14.4.1 Creating a project file

This discussion assumes that you are familiar with your preferred IDE, that you know how to create and configure a project file, and that you have a project configured to work with each CPU on a multi-CPU PSoC™ 6 MCU device. See [Appendix A](#) for information on the kinds of options that must be configured.

You can start with a new, empty project file. However, Infineon provides a pre-configured template project for each supported IDE and PSoC™ 6 MCU CPU. There are CM4 and CM0+ projects for the µVision IDE, the IAR IDE, and the other supported IDEs. PDL template projects are fully described in the [PDL v3.1 User Guide](#) section titled Using PDL Template Projects. Template projects are located here: <PDL Install Folder>/devices/psoc6/templates.

Most build options in the template project are set correctly. Changes required when importing the generated code are discussed in this application note. In addition, the include paths in the template are project-relative. If you move the template project file to a different location, you must modify these paths. See [Where to get PDL library files](#).

5.14.4.2 Locating and identifying source files

PSoC™ Creator generates three kinds of source files:

- User files – files you may modify
- Design files – files specific to the PSoC™ Creator design and Components
- Library files – files from the PDL

The following sections describe each kind of source file in detail.

PSoC™ Creator puts these files in various folders inside the <project>.cydsn folder. This application note refers to this location simply as the cydsn folder. [Figure 524](#) shows the folder tree and locations for generated source files.

5 PSoC™ 6 application notes

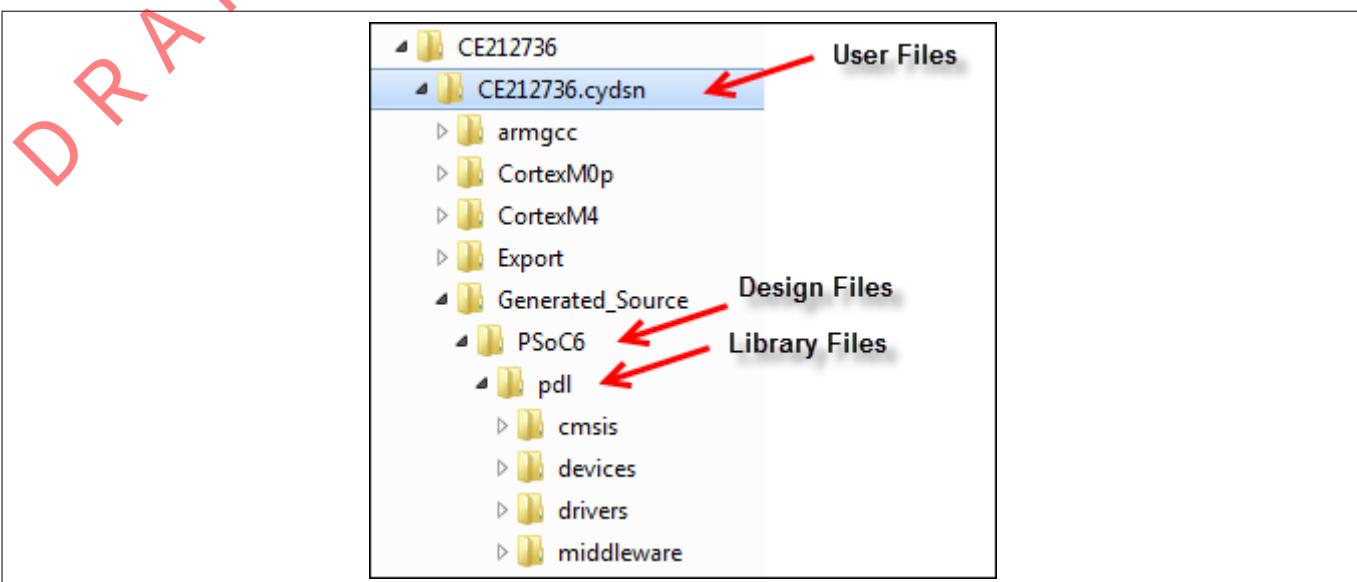


Figure 524 Where to find generated file

5.14.4.2.1 User files

A user file is one that you may change based on the requirements of your design. PSoC™ Creator generates the file (which may be functionally empty), and never changes it. These are the key concepts to understand about user files:

- The files are created automatically when you create a PSoC™ Creator project and generate the application for the first time
- PSoC™ Creator does not change or replace these files once they are created. Therefore, you can modify these files as required for your design
- All files are local copies, so changes have no impact on other projects or the installed PDL library
- For a multi-CPU device, add CPU-specific files to the correct project in your IDE; add shared files to the project for each CPU

You do not need every file; some are specific to a particular tool chain.

[Table 114](#) lists common user files. User files are at the top level of the cydsn folder (except for the startup code, as noted in the table). Some file names vary based on the device. The table uses the psoc63 series as the example. Some files are CPU-specific. Startup code and linker files are also IDE-specific. Use the appropriate files for your project. Disregard the others. If you use an unsupported IDE, provide equivalent files compatible with the IDE.

Depending upon your design, there may be additional user files. Any source file at the top level of the cydsn folder is a user file and should be added to one or both projects, depending on whether the file is CPU-specific.

Table 114 PSoC™ Creator generated user files

File	Action	Source	Usage	Notes
main_cm0p.cmain_cm4.c	Add to project ³⁷⁾ or provide your own.	Generated	Required	PSoC™ Creator puts the user files at the top level of the cydsn folder

(table continues...)

³⁷ Add CPU-specific files to the project for the corresponding CPU.

5 PSoC™ 6 application notes

Table 114 (continued) PSoC™ Creator generated user files

File	Action	Source	Usage	Notes
cyapicallbacks.h	Provide the path to the header.	Generated	Required	An empty file for the user to implement macro callbacks (see PSoC™ Creator Help topic Writing Code)
system_psoc6.h	Provide the path to the header.	Copied from PDL	Required	Contains user-definable clock configuration macros
system_psoc6_cm4.c system_psoc6_cm0plus.c	Add to the project.	Copied from PDL	Required	Per-CPU; system configuration code
startup_psoc6_01_cm4.s startup_psoc6_01_cm0plus.s (IAR)	Add to the project. For an unsupported IDE, provide a startup file.	Copied from PDL	Use the file for your IDE	Per-series, CPU, and IDE; startup code is located in .cydsn/<IDE>/startup
startup_psoc6_01_cm4.s startup_psoc6_01_cm0plus.s (MDK)				
startup_psoc6_01_cm4.S startup_psoc6_01_cm0plus.S (GCC)				
cy8c6xx7_cm4_dual.icf cy8c6xx7_cm0plus.icf	Set project options to use the CPU-specific linker file. For an unsupported IDE, provide a linker file.	Copied from PDL	Use the file for your IDE	Per-series, CPU, and IDE; linker command file
cy8c6xx7_cm4_dual.scat cy8c6xx7_cm0plus.scat				
cy8c6xx7_cm4_dual.ld cy8c6xx7_cm0plus.ld				

Note: All header files listed in Table 114 are in the same folder, so a single path provides access to all.

Note: Although the default cyapicallbacks.h file is empty, there is no easy way to remove it from the project. It is included by another automatically generated header file. If you delete the #include statement that includes this header, that statement reappears the next time you generate code.

Note: A PDL template project has user files and paths from the installed PDL location, rather than the generated files (copies) in the cydsn folder. When manually importing code, replace the files with the local copies in the cydsn folder. See [Where to get PDL user files](#).

While some files are CPU-specific, some are used by each CPU in a multi-CPU device. The file system_psoc6.h is a good example. Each CPU uses the same file to set up CPU clocks based on the user-definable clock configuration macros. When manually importing a shared file into an IDE, the file must be added to the project for each CPU. If in doubt, right-click a file in the PSoC™ Creator Workspace Explorer pane and examine its properties to see which CPU toolchain uses the file.

5 PSoC™ 6 application notes

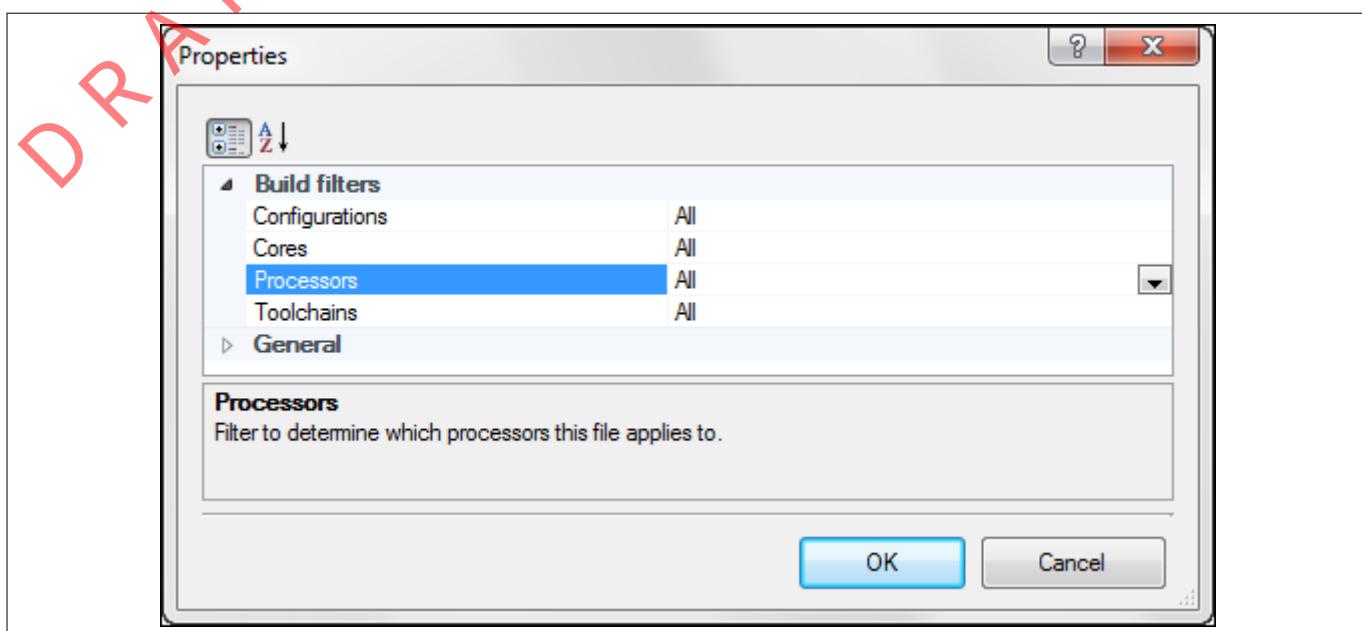


Figure 525 Properties for the system_psoc6.h file

5.14.4.2.2 Design files

PSoC™ Creator automatically generates several files that are specific to your design. In some cases, these files are specific to a supported IDE as well. All design files are in this folder: cydsn/Generated_Source/PSoC6.

Design files are described in PSoC™ Creator Help in the Building a Creator Project > Generated Files (PSoC 6) topic. The design files also include a source and header file for each Component in the design. If a Component is based on a Universal Digital Block (UDB), all the files required to configure and use the Component are among the design files.

These are the key concepts to understand about design files:

- PSoC™ Creator may change these files any time you generate the application
- Do not modify these files; any change will be lost when you generate the application
- Component-specific files are named <Component Name>.h and <Component Name>.c
- Some files are optional and/or IDE-specific (see [Table 115](#))

Most design files are shared files. For a multi-CPU device, add shared files to the project for each CPU. Any CPU-specific file identifies the CPU in the file name.

[Table 115](#) lists design files, and the action required to use a file in your project

Table 115 PSoC™ Creator generated design files

File	Action	Usage	Notes
project.h	Provide path to header ³⁸ .	Required	Includes all headers listed in this table
<Component Name>.c/h	Add source file to project. Provide path to header.	Required	Code for component-specific APIs. Header file included by project.h
cyfitter.h	Provide path to header.	Required	#define for design-specific addresses and values

(table continues...)

³⁸ All header files listed in this table are in the same folder, so a single path provides access to all.

5 PSoC™ 6 application notes

DRAFT Table 115 (continued) PSoC™ Creator generated design files

File	Action	Usage	Notes
cyfitter_cfg.c/h	Add source file to the project. Provide path to header.	Required	Code to configure the device before reaching main()
cyfitter_gpio.h	Provide path to header.	Required	#define for design-specific addresses and values for pin configuration
cyfitter_sysint.h	Provide path to header.	Required	#define for design-specific interrupts
cyfitter_sysint_cfg.c/h	Add source file to project. Provide path to header.	Required	Code to configure system interrupts
cymetadata.c	Add to project.	Required	Defines all extra memory spaces that need to be included.
cydevice_trm.h	Provide path to header.	Required	Defines all addresses in the configuration space of the device
cydisabledsheets.h	Provide path to header.	Required	#define for disabled schematic pages; empty if there are none
cydevicegnu_trm.inc	Include in your assembler code if required.	Optional	Per IDE; defines all addresses in the configuration space of the device for the assembler
cydeviceiar_trm.inc			
cydevicerv_trm.inc			
cycodeshareexport.ld	Set project options to use the script if required.	Optional	Per-IDE; linker script to export or import symbols to/from a different application; typically, empty
cycodeshareimport.ld			
cycodeshareimport.scat			

The default `main_cm0p.c` and `main_cm4.c` files include `project.h`. The `project.h` file in turn includes all other required headers among the design files. If you provide your own `main.c` files, ensure that you include `project.h`.

Although `cydisabledsheets.h` is typically empty, there is no easy way to remove it from the project. It is included by `project.h`. If you delete the `#include` statement, that statement reappears the next time you generate code.

5.14.4.2.3 Library files

Library files are copies of files from the PDL installation. PSoC™ Creator automatically copies the required files into subfolders of the `cydsn/Generated_Source/PSoC6/pd1` folder. See the PDL v3.1 User Guide to learn about the PDL.

These are the key concepts to understand about library files:

- Only required files are copied, not the entire PDL
- Some files may be assembler source files, or precompiled binaries
- PSoC™ Creator never modifies library files
- Do not modify these files; any change will be lost when you generate the application
- Most library files are shared files. For a multi-CPU device, add shared files to the project for each CPU. Any CPU-specific file identifies the CPU in the file name

What library files appear in the `pd1` folder tree depends upon your design. Table 116 lists the key locations in the tree, what is in each folder, and how to use those files in a project.

~~5 PSoC™ 6 application notes~~

Table 116 PSoC™ Creator generated library files

Path/Folder	Action	Notes
pdl/cmsis/include	Provide a path to this folder.	CMSIS header files
pdl/devices/psoc6/include/ip	Provide a path to this folder.	Header files for IP blocks on the device
pdl/devices/psoc6/include	Provide a path to this folder.	Device-specific header files
pdl/drivers/peripheral	Add source files to project. Provide a path to this folder and all the subfolders.	Source and header files for peripheral drivers used in the design
pdl/middleware	Add source files to project. Provide a path to this folder.	Source and header files for the middleware used in the project; do not add paths to subfolders.

Note: *The system library (syslib) peripheral driver has an assembler file as part of its implementation. It must be added to the project along with the C source file. [Manually importing generated code – an example](#) shows you how.*

Note: *The pdl/middleware folders might have subfolders. Do not add paths to each subfolder. The PDL source code provides the path in the #include statement.*

Note: *A PDL template project has paths that point to the installed PDL location, rather than the generated files (copies) in the cydsn folder. When manually importing files, reset these paths to point to the cydsn folder. See [Where to get PDL library files](#).*

5.14.4.3 Adding files to a project

As you identify the files you need, add them to the IDE's project file. Each IDE has its own way of adding a file to a project. For example, the IDE may support drag and drop, or adding a batch of files through the IDE's user interface.

There are three key principles to keep in mind as you add the files:

First, add each file directly from its location in the PSoC™ Creator cydsn folder. This is critical to support iterative development. If you change the design in PSoC™ Creator, some generated files will change. By pointing the IDE project at the files in the cydsn folder, you ensure that any changes to the files appear in your project.

Second, some files are CPU-specific. Add CPU-specific files to the corresponding project. For a multi-CPU device, add shared files to both projects. The file name for a CPU-specific file identifies the CPU.

Third, you must add include paths so that the preprocessor can locate required header files. The good news is that most headers are grouped into single folders. Some developers prefer to add the header files to a project for easy access to the file. Even if you do, most IDEs still require that you add include paths for the preprocessor.

5.14.4.3.1 Where to get PDL user files

Some PSoC™ Creator generated user files start as copies of default PDL files. Do not use the original PDL files. Use the generated copies in the cydsn folder.

5 PSoC™ 6 application notes

User files may be changed by the user. If you change the original PDL file, that change affects any other project using that file. Best practice dictates that you use the copies in the cydsn folder. See [User files](#).

If you start with a PDL template project, you should:

- Remove the user files from the project (they are the originals from the PDL installation)
- Add the same files back to the project from the cydsn folder
- Delete the project-relative path to the /include folder. It typically looks something like this in an IDE: ../../include
- Add a path to the cydsn folder where PSoC™ Creator puts the user files

If you start with an empty project, add the files from the cydsn folder, and set a path to this folder.

5.14.4.3.2 Where to get PDL library files

Library files are copies of the PDL files and should never be changed by the user. Add these files from the cydsn/Generated_Source/PSoC6/pd1 folder. PSoC™ Creator copies only the library files you need.

A PDL template project has preset paths to PDL header files that point to original files in the PDL installation, not the generated code. If you start with a PDL template project, you should:

- Delete any project-relative path to the PDL installation
- Replace with paths to the same locations in the cydsn/Generated_Source/PSoC6/pd1 folder

The preset paths may include /cmsis/include, /ip, /drivers/peripheral etc.

5.14.4.4 Supporting iterative development

After you import the generated code and set project options so that the project builds in the IDE, you will encounter the need to change your design. Importing files into an IDE involves some effort, but this is a one-time task.

The key requirement to support iterative development is simple: import the files from the cydsn folder. After making design changes in the PSoC™ Creator project, build the code. Depending on the changes, the code generation process may modify, add, or remove files.

Modified files

PSoC™ Creator modifies only design files. If you add the generated code from the cydsn folder to your project file, the next build will use the latest version of the files. Rebuild your code in the IDE to update the executable. No additional work is required.

Added or removed files

Add any new file (either design file or library file) to your project. Similarly, if a file is no longer necessary, you can remove it from your project file. Then rebuild your code in the IDE to update the executable.

5.14.4.5 Manual import review

While this section provides significant details about PSoC™ Creator generated code, the process of manually importing generated code into an IDE project file is quite simple:

1. Identify the generated files you need (user, design, and library)
2. Add them to the project file (from their location in the cydsn folder)
3. Set include paths to find the header files

~~DO NOT USE~~

5 PSoC™ 6 application notes

5.14.5 Manually importing generated code – an example

This section walks you through the process of importing generated code from a relatively complex code example. Because every project and IDE is different, this example is not a series of precise steps and directions.

To gain maximum value from this walk-through, you should download, install, and build (using PSoC™ Creator) the [CE21736 code example](#). Create a new empty project in an IDE. Then explore the contents of the PSoC™ Creator cydsn folder as you import the code.

A single example cannot cover all eventualities. Among the variables are the choice of project file (template or empty), which IDE to use (many possibilities), the nature of the firmware itself, and the particulars of your build environment.

The information in [Manually importing generated code](#) should enable you to make informed decisions for your circumstances. Given the broad number of choices, each of which would result in a different example, this walk-through is based on the following options:

- An empty project file – This approach supports any IDE
- Keil µVision tools – Although this is a supported IDE, this choice provides a contrast to allow users to determine which path is better for their circumstances, CMSIS Pack or manual import
- CE212736 as the example project – because it provides a rich domain for exploring the choices you face when importing code
- CE212736 – PSoC™ 6 MCU with Bluetooth® Low Energy Connectivity uses the Bluetooth® Low Energy stack, assembler source code, and precompiled binary files, as well as several peripheral drivers. This walk-through does not run the code example on hardware or explain its functionality

This walk-through imports code into a CM4 project. For a multi-CPU device, you build both CM0+ and CM4 projects.

After you complete the design and build the PSoC™ Creator project, the process is identical for each CPU:

1. Set up the IDE project file
2. Add files to the project
3. Set include paths
4. Build the project in the IDE

5.14.5.1 Complete the design and generate application files

You must be able to successfully generate the application in PSoC™ Creator before you import to an IDE.

[Figure 526](#) shows the PSoC™ Creator workspace explorer after generating the application files for the CE212736 project.

5 PSoC™ 6 application notes

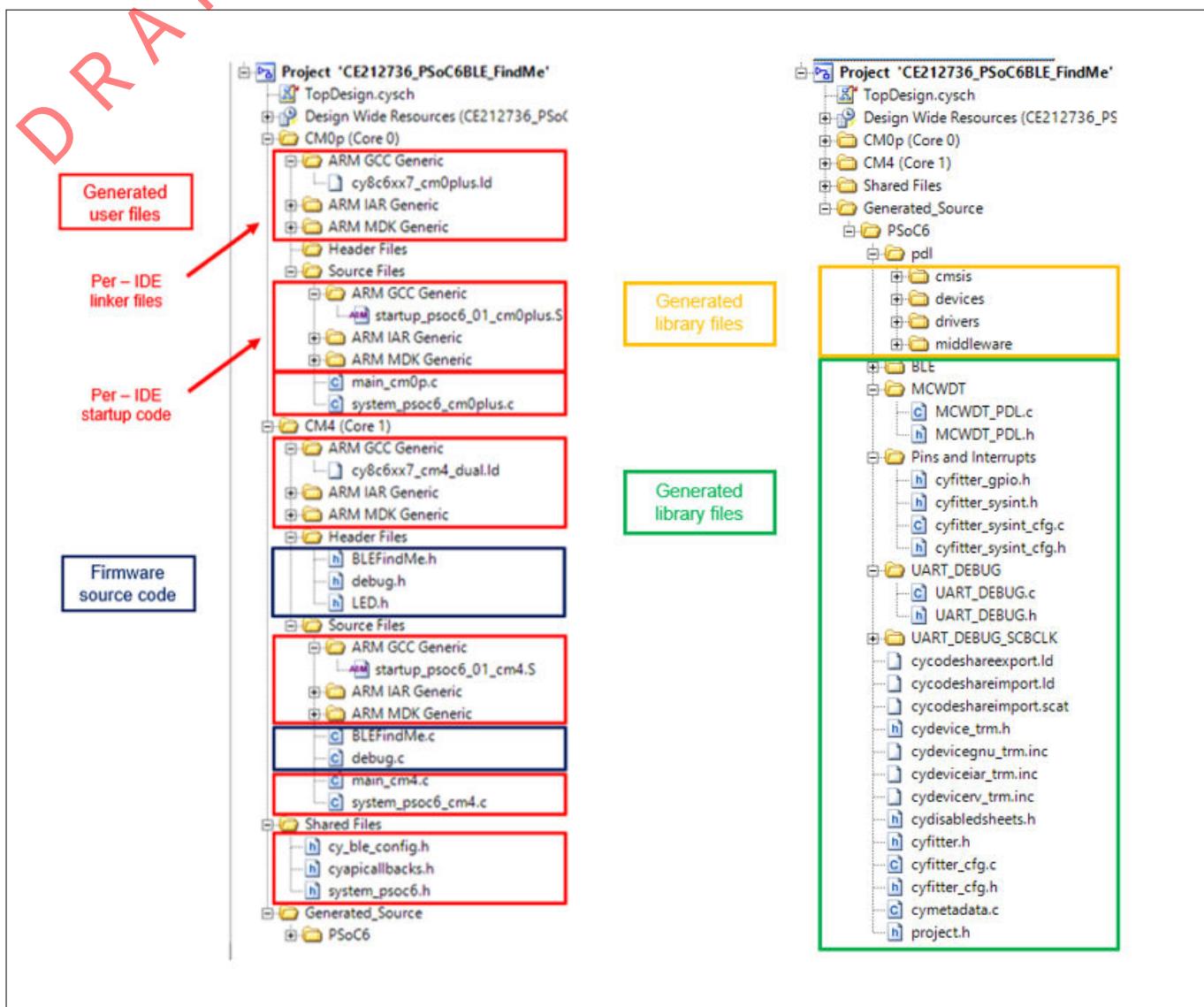


Figure 526 PSoC™ Creator workspace explorer

The Generated_Source item in the Workspace Explorer pane contains all the files that are in the project's cydsn/Generated_Source folder. This design uses a Bluetooth® Low Energy Component, a multi-counter watchdog timer (MCWDT), and a UART to print debug messages to a terminal window. It also has pins to control LEDs and sets up system interrupts.

Because this is a complete code example, the project also contains the firmware source code to implement the example. If you prefer to develop code in another IDE, typically you would create these files in the IDE, not in PSoC™ Creator.

5.14.5.2 Set up the IDE project file

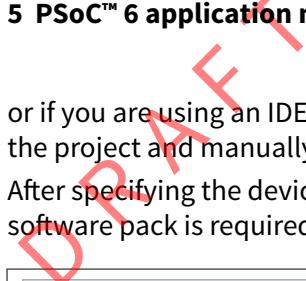
Create a new project. Typically, the IDE requires that you specify the target device. There are likely to be additional steps in an IDE's project creation process.

For example, in Keil µVision tools, choose **Project > New µVision Project**. Specify the target device. You can download the PSoC™6_DFP pack from the [Infineon GitHub repository](#) and install the pack. The pack provides PSoC™ 6 device support in Keil µVision and other IDEs that support CMSIS packs. Using the right device will automatically set the memory address range, programming algorithm, and so on. If the device is not available,

5 PSoC™ 6 application notes

or if you are using an IDE that does not support CMSIS packs, you can start by selecting the right Arm® CPU for the project and manually provide the required details specific to the device.

After specifying the device, pick software packs. Figure 527 shows an example. For this walk-through, no software pack is required.



Software Component	Sel.	Variant	Version	Description
+ CMSIS				Cortex Microcontroller Software Interface Components
+ CMSIS Driver				Unified Device Drivers compliant to CMSIS-Driver Specifications
+ Compiler		ARM Compiler	1.1.0	Compiler Extensions for ARM Compiler ARMCC and ARMClang
+ Device				Startup, System Setup
+ File System		MDK-Pro	6.9.0	File Access on various storage devices
+ Graphics		MDK-Pro	5.36.6	User Interface on graphical LCD displays
+ Network		MDK-Pro	7.3.0	IPv4/IPv6 Networking using Ethernet or Serial protocols
+ USB		MDK-Pro	6.9.0	USB Communication with various device classes

Figure 527 μVision software pack selection

When done, an empty project appears, as shown in Figure 528. For this example, we named the project as "CM4 Project". When importing code for a multi-CPU device, you need a project for each CPU.

5 PSoC™ 6 application notes

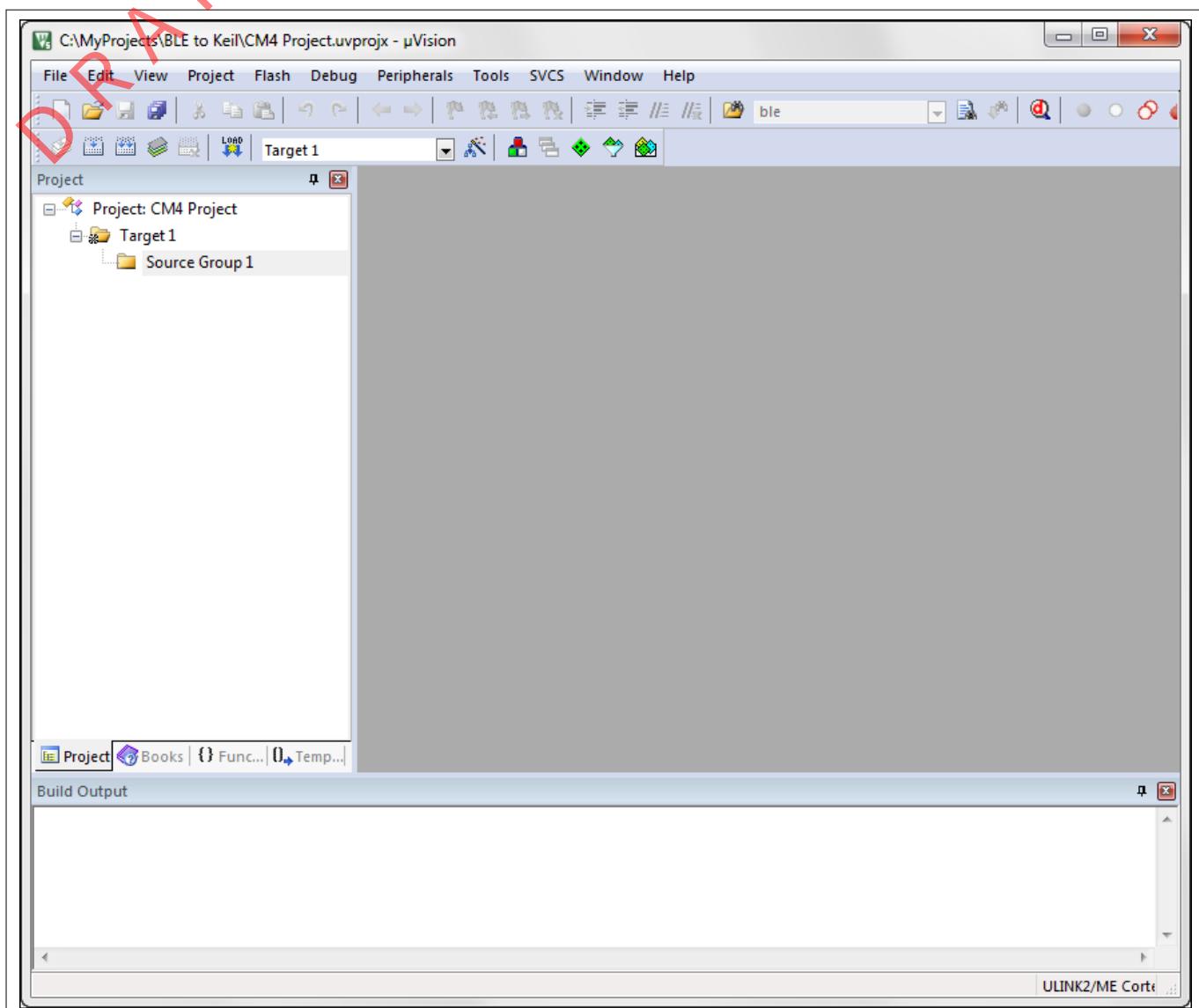


Figure 528 An empty project in the μVision IDE

5.14.5.3 Add files to the project

Identify the files you need and add them to your project. Each IDE has its own workflow for adding files. An IDE typically has a file explorer pane of some kind. You can add or remove groups within the project to organize your files as you see fit. The guidance in this section is merely that: guidance. You may prefer another organizational scheme. [Figure 529](#) shows the group structure used in this walk-through. It matches the kinds of files PSoC™ Creator generates, with one exception. Because the Bluetooth® Low Energy stack is a large collection of files, this example puts them in a dedicated group.

5 PSoC™ 6 application notes

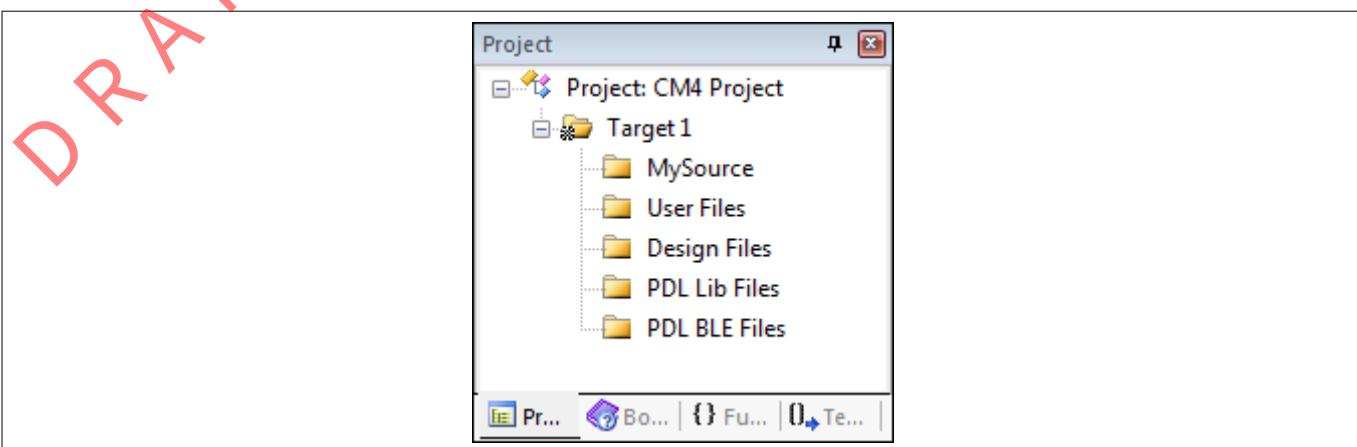


Figure 529 Arbitrary group structure in the μVision IDE

5.14.5.3.1 Add firmware files

Because this walk-through imports a fully-functional code example, we import the firmware files into the MySource group, as shown in [Figure 530](#). The firmware files are all in the cydsn folder for the PSoC™ Creator project. In your own work, you would create your own firmware files, and organize them in the project file as you see fit.

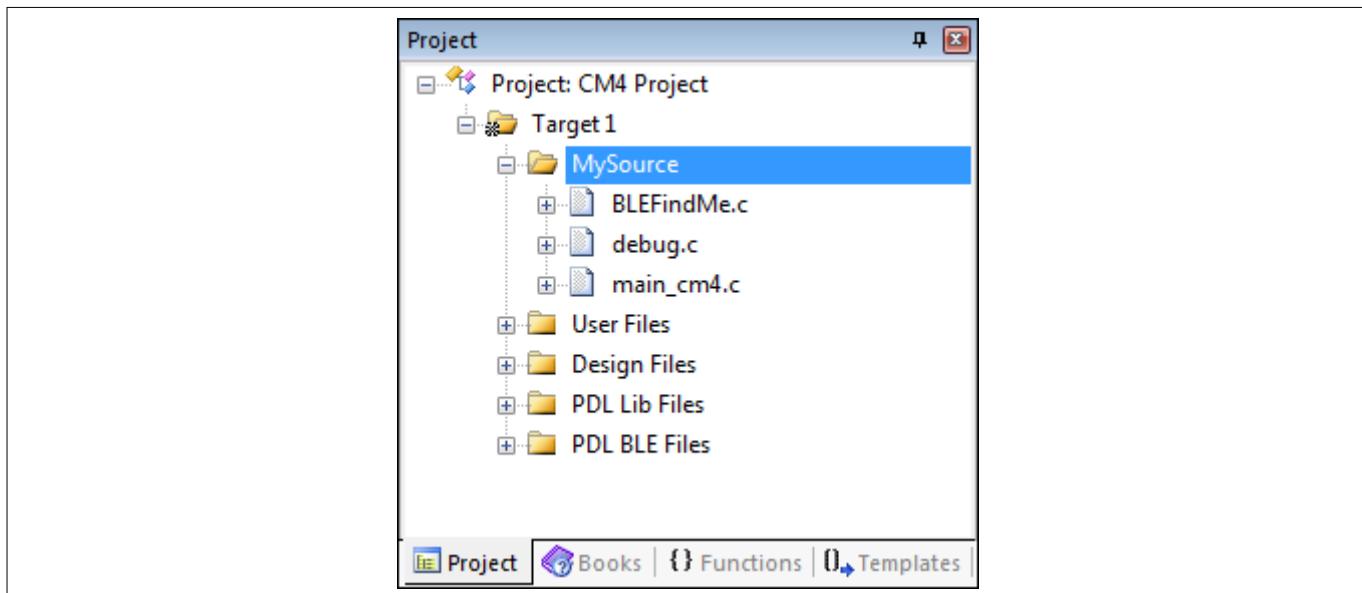


Figure 530 Firmware files added to the project

Note: When repeating this process for the CM0+ project, add files specific to that CPU.

5.14.5.3.2 Add user files

User files are generated by PSoC™ Creator, but you can change these files. User files are located at the top level of the cydsn folder. The startup code is in an IDE-specific folder. See [User files](#). In this case, the user files are:

- A system configuration file (CPU-specific)
- A startup file (IDE- and CPU-specific)

[Figure 531](#) shows the user files in the project explorer.

5 PSoC™ 6 application notes

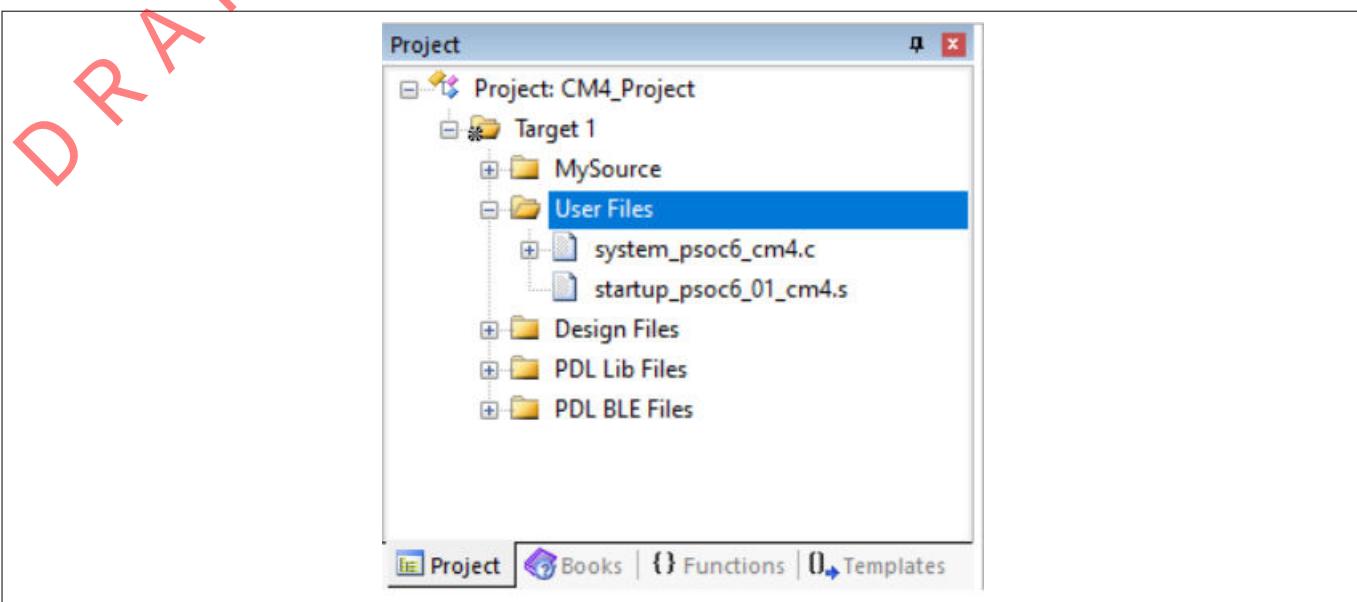


Figure 531 User files added to the project

Note: *The startup file is IDE-specific assembler source code. The generated code includes the startup code for all supported IDEs. Use the file you need. If you are working with an unsupported IDE, provide your own startup code.*

5.14.5.3.3 Add design files

Design files are the source and header files generated by PSoC™ Creator specific to your system design and its Components. See [Design files](#). Design files are in the cydsn/Generated_Source/PSoC6 folder.

Figure 532 shows the design files for this project added to the µVision project. These include the cyfitter and cymetadata files, as well as the Component-specific source files for the Bluetooth® Low Energy, MCWDT, and UART Components.

5 PSoC™ 6 application notes

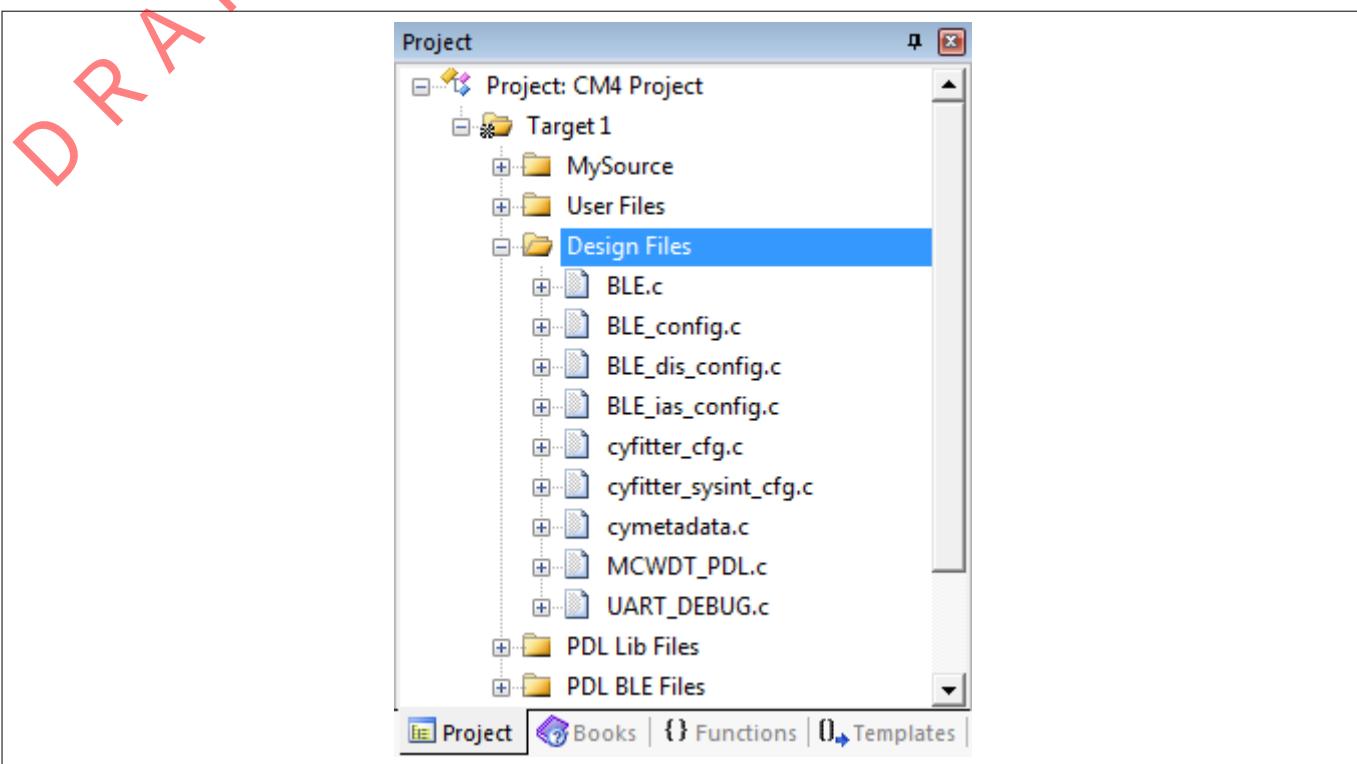


Figure 532 Design files added to the project

5.14.5.3.4 Add PDL library files

Library files are copied from the PDL installation. See [Library files](#). Based on your design, PSoC™ Creator copies required files into the cydsn/Generated_Source/PSoC6/pdl/drivers/peripheral folder. [Figure 532](#) shows the peripheral driver files for this project added to the IDE.

5 PSoC™ 6 application notes

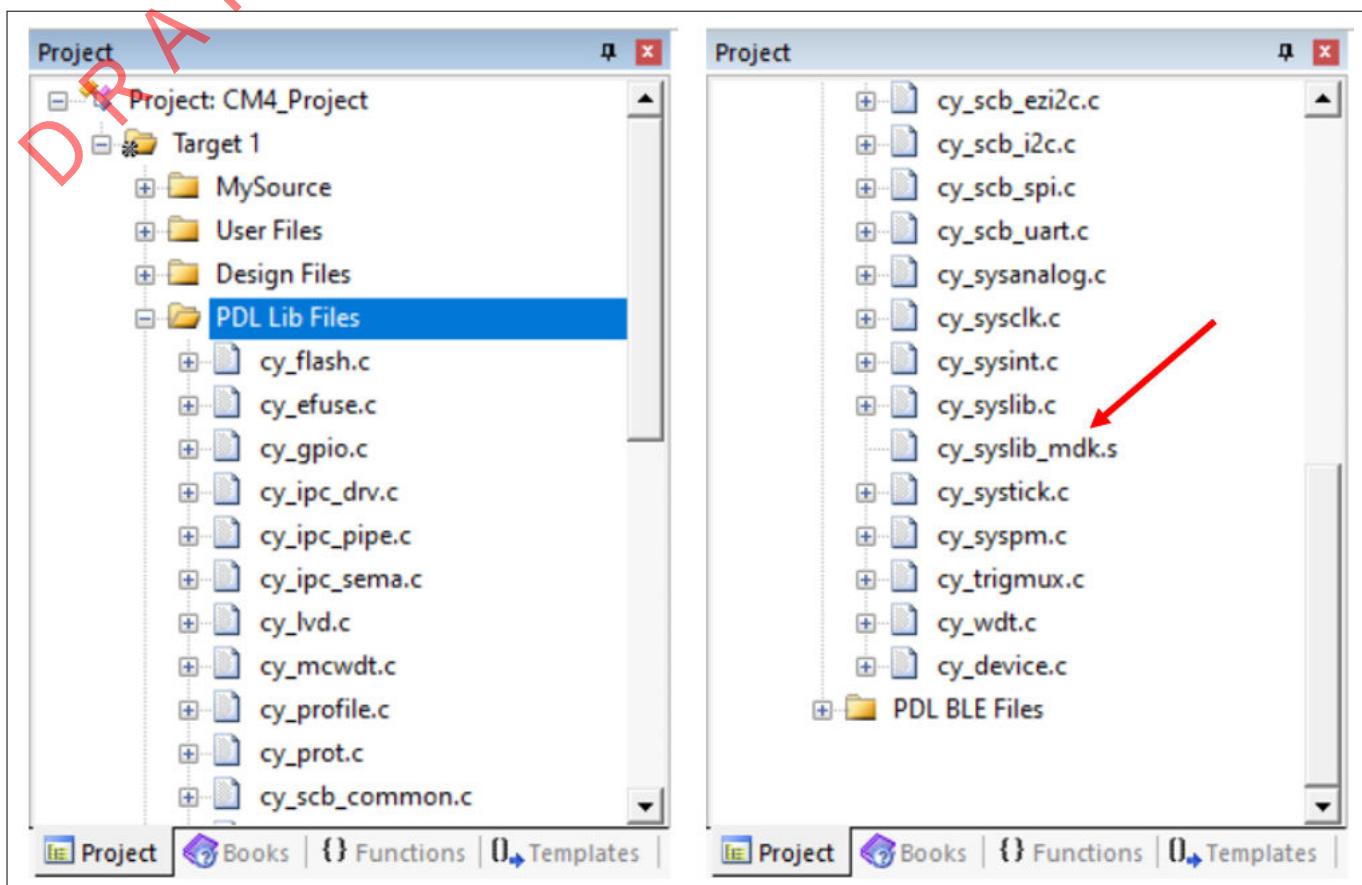


Figure 533 Library files added to the project

Note: The implementation for the system library (syslib) includes an IDE-specific assembler file, called out in [Figure 533](#). If this file is not added to the project, the build fails.

Note: Each driver has its own subfolder within the peripheral folder. Adding source files can be tedious as you navigate up and down the folder structure. If an IDE supports adding files by drag and drop, you can use Windows Explorer to make the task easier. Go to the peripheral folder, and search for *.c. All the .c files appear as shown in [Figure 534](#). You can then drag these into the IDE. (Unfortunately, this tip does not work for the µVision IDE.) Don't forget to add the assembly source file as well.

5 PSoC™ 6 application notes

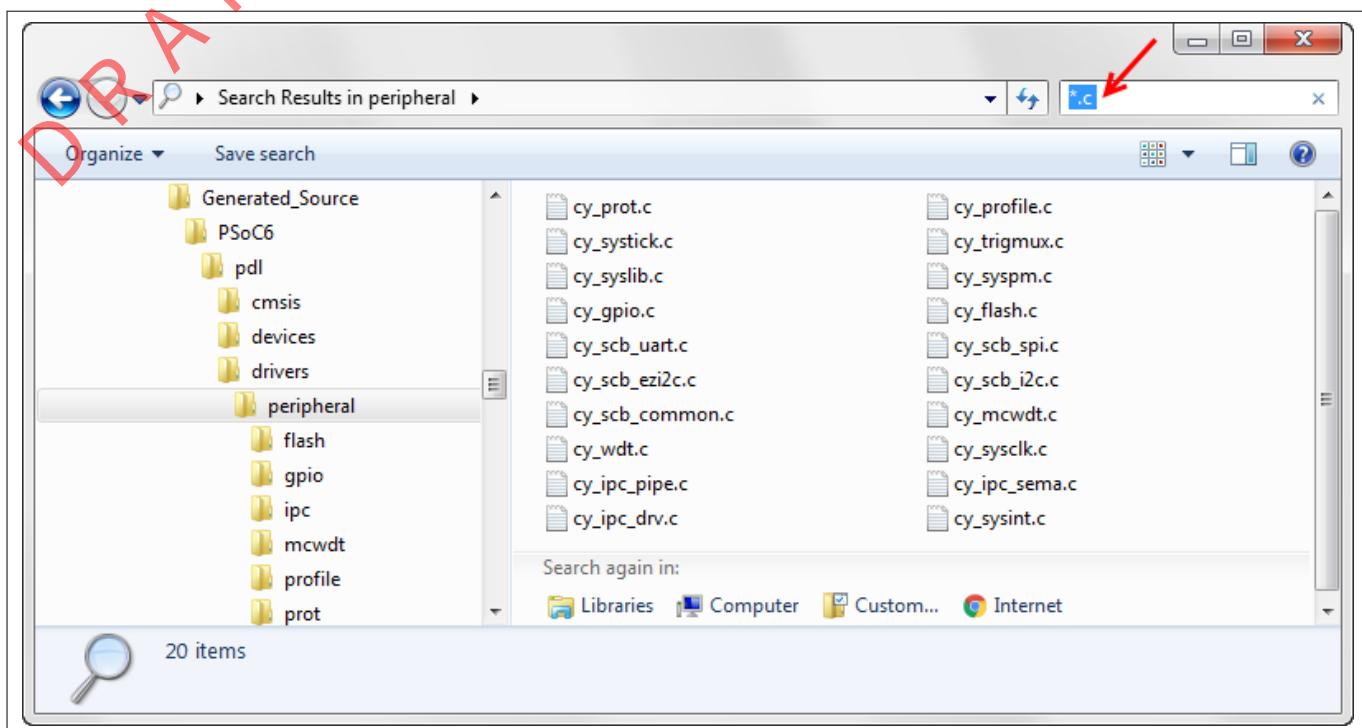


Figure 534 Searching for .c files

5.14.5.3.5 Add Bluetooth® Low Energy library files

The final collection of files used in this design is the Bluetooth® Low Energy stack. The generated code is located at `cydsn/Generated_Source/PSoC6/pdl/middleware/ble`. Some parts of the Bluetooth® Low Energy stack are provided as CPU-specific binary libraries, not source code. All required source files and binary libraries must be added to the project.

5 PSoC™ 6 application notes

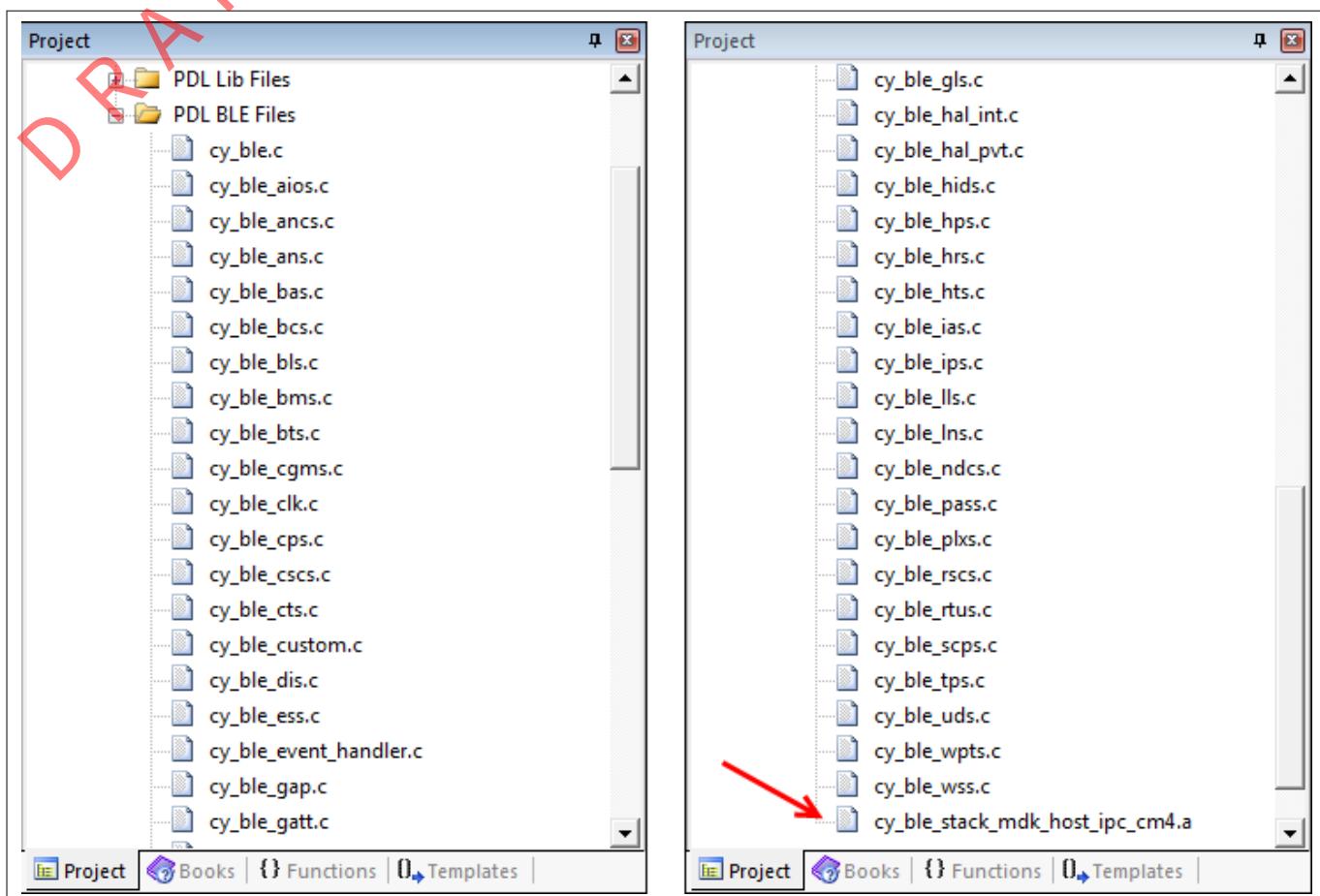


Figure 535 **Bluetooth® Low Energy stack files added to the project**

Note: When repeating this process for the CM0+ project, add libraries specific to that CPU.

Note: An IDE may treat the library file as an assembler source file rather than a library, which causes build errors. The IDE may have a property to control this. For example, Figure 536 shows the file properties dialog for the µVision IDE.

5 PSoC™ 6 application notes

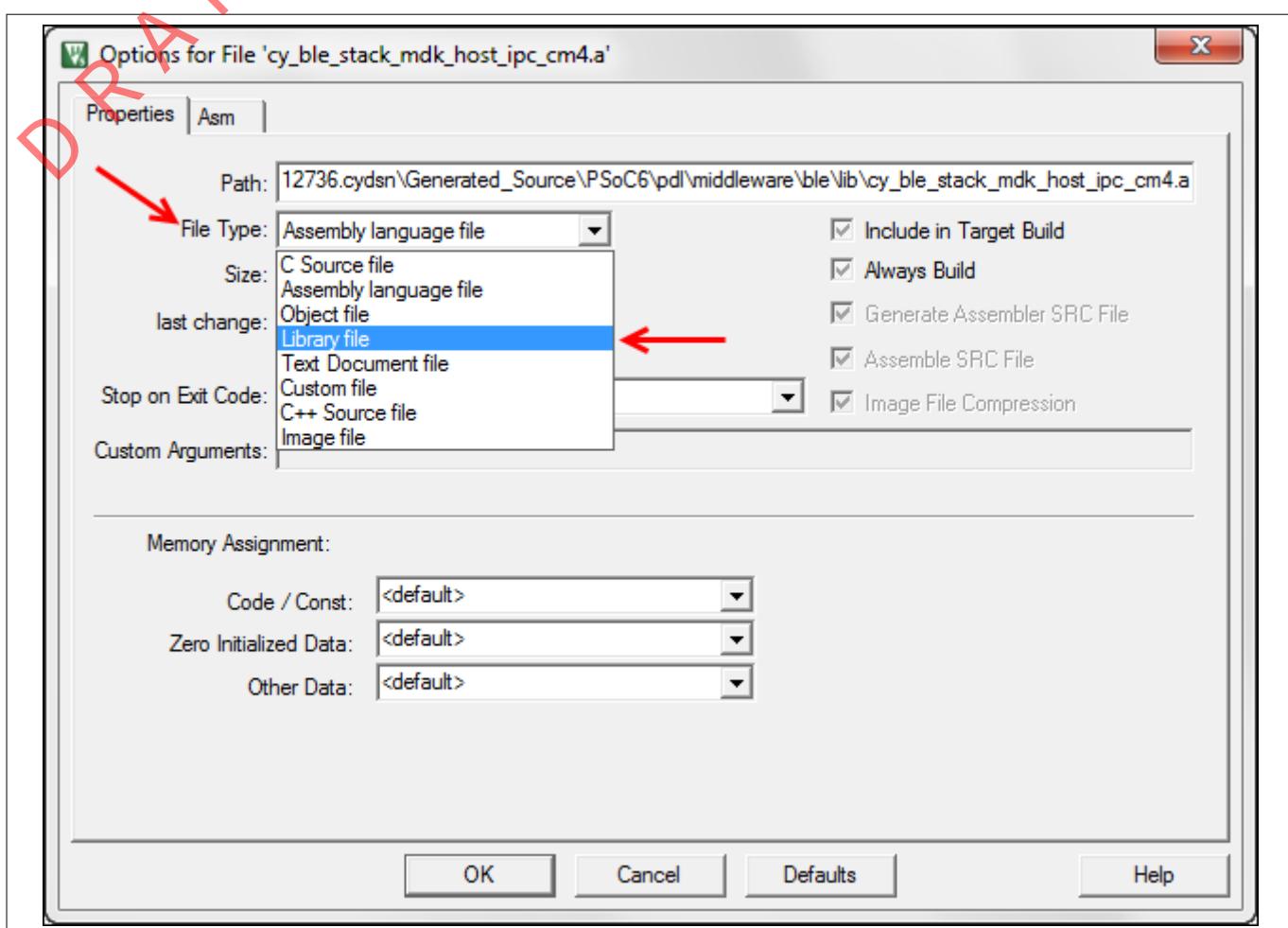


Figure 536 Setting the file type in the μVision IDE

5.14.5.4 Set include paths

The IDE's build system must locate all necessary header files. Some developers prefer to add header files directly to the project explorer for ease of access. However, this typically does not set include paths for the IDE. Each IDE has its own way of setting include paths. This example assumes that you know how to add include paths in your preferred IDE. The goal of this section is to show you what paths you need to add, not how to add them.

In the μVision IDE, you add paths in the target options C/C++ panel in the **Include Paths** item.

5.14.5.4.1 Set a path to user files

All header files for the PSoC™ Creator generated user files are in the cydsn folder. The path of course depends upon where you put the PSoC™ Creator project. In this example, the path for the CE212736 project could look like this:

C:/MyProjects/CE212736/CE212736.cydsn

Figure 537 shows setting that path in the μVision IDE.

5 PSoC™ 6 application notes

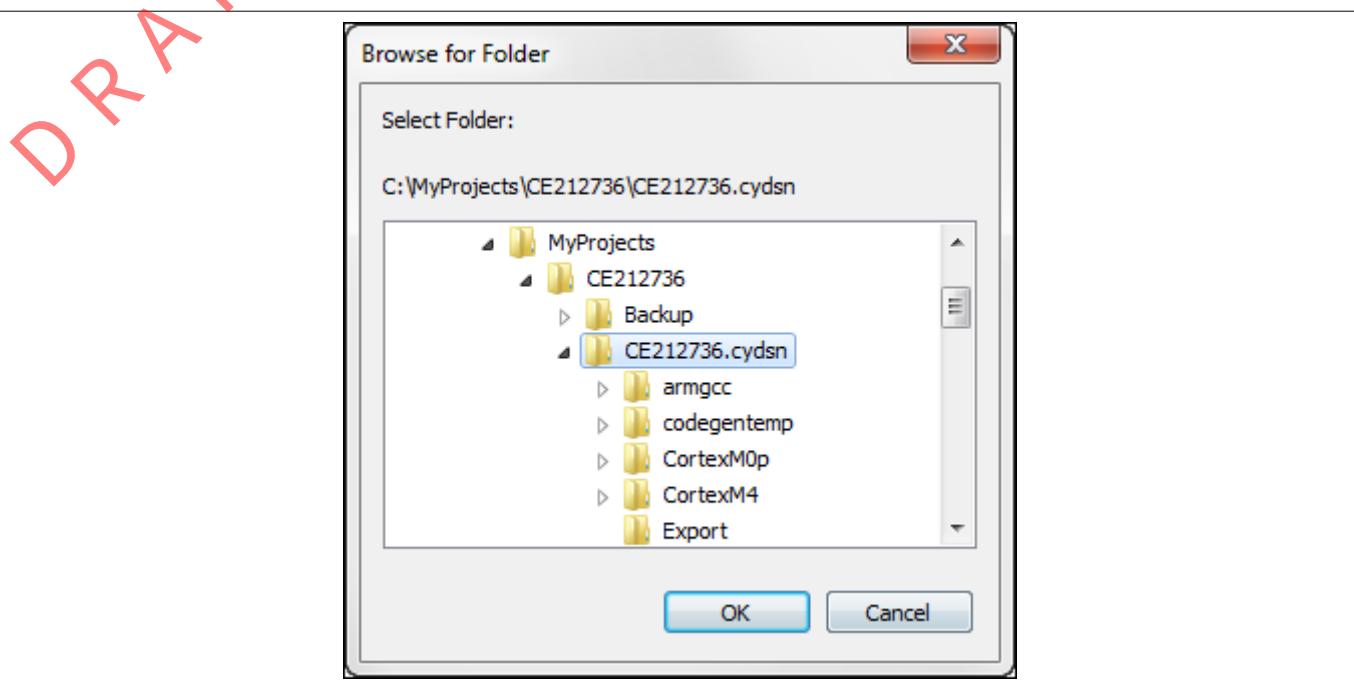


Figure 537 Setting the path to the cydsn folder

Note: *In this example, the firmware header files are also in this folder, so there is no need for a separate path for them.*

5.14.5.4.2 Set a path to design files

All header files for the PSoC™ Creator generated design files are in the cydsn/Generated_Source/PSoC6 folder. In this example, the path for the CE212736 project could look like this:

C:/MyProjects/CE212736/CE212736.cydsn/Generated_Source/PSoC6

Figure 538 shows setting that path in the µVision IDE.

5 PSoC™ 6 application notes

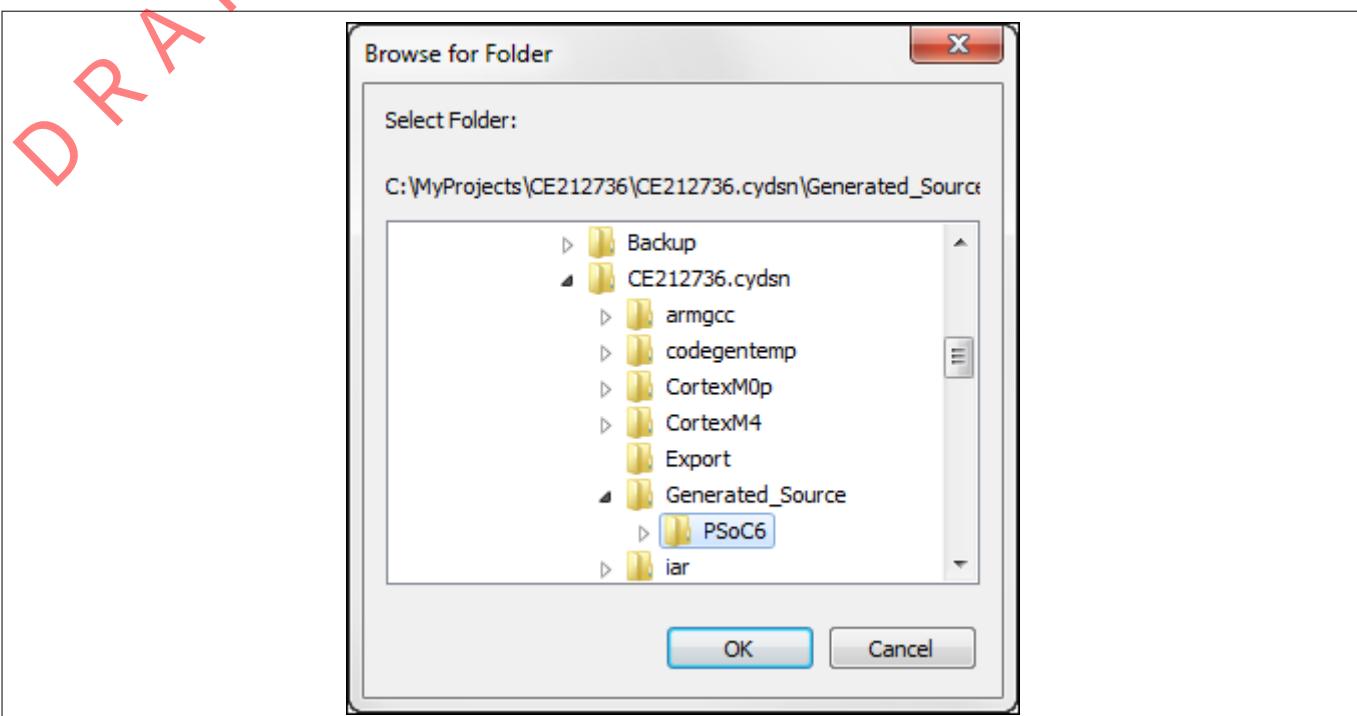


Figure 538 Setting the path to the design files

5.14.5.4.3 Set a path to peripheral folder (driver files)

Each peripheral driver has its own subfolder, and each peripheral driver has a header file that needs to be included in the project. So, the path to each driver subfolder needs to be added as part of the Include Paths. For example, the path to be added to include the flash driver header file in this project would be:

C:/MyProjects/CE212736/CE212736.cydsn/Generated_Source/PSoC6/pd1/drivers/peripheral/flash

Along with this, the path to the peripheral folder that contains all driver subfolders should also be added. The peripheral folder path for the CE212736 project would be:

C:/MyProjects/CE212736/CE212736.cydsn/Generated_Source/PSoC6/pd1/drivers/peripheral

Figure 539 shows the setting of the peripheral folder path in the μVision IDE.

5 PSoC™ 6 application notes

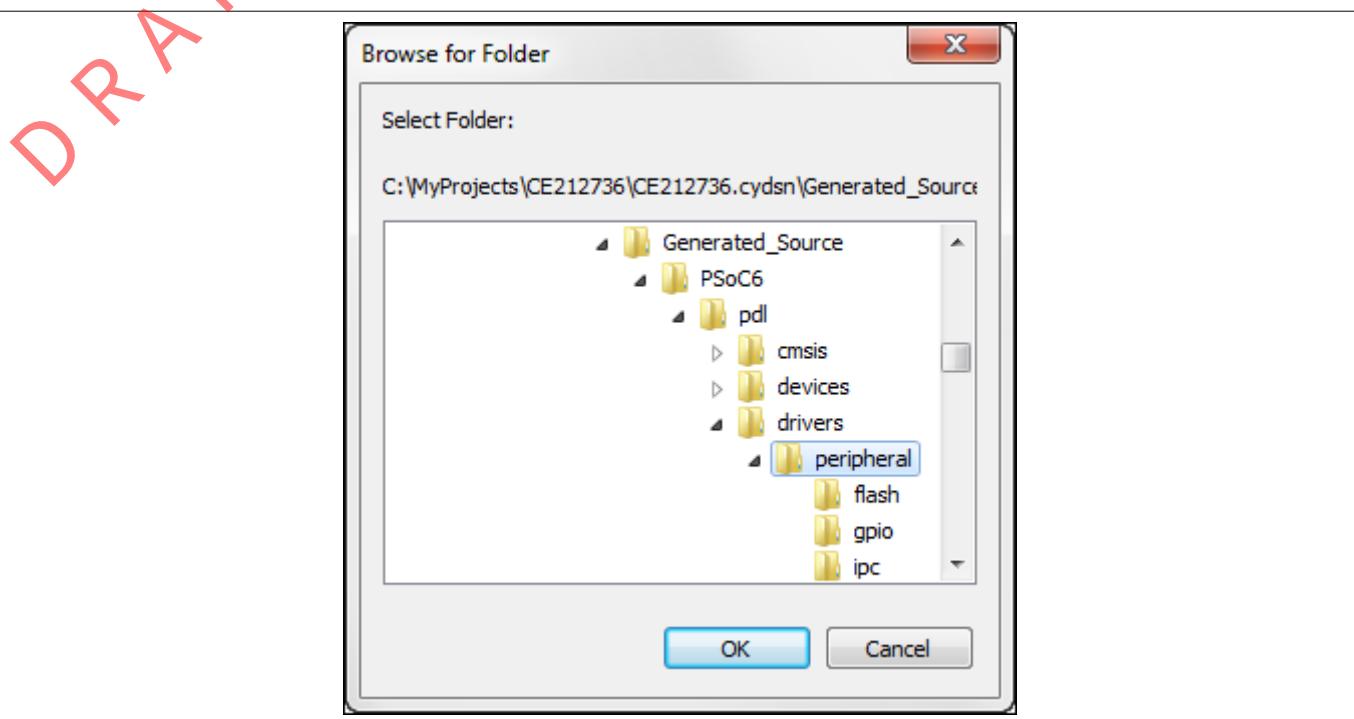


Figure 539 Setting the path to the peripheral folder

5.14.5.4.4 Set a path to the `middleware` folder (Bluetooth® Low Energy stack)

Although the Bluetooth® Low Energy stack is in a subfolder, the source code provides the path to include any header file within the middleware folder. As a result, a single path works for all locations in this folder tree. In this example, the path for the CE212736 project could look like this:

C:/MyProjects/CE212736/CE212736.cydsn/Generated_Source/PSoC6/pdl/middleware

Figure 540 shows setting that path in the μVision IDE.

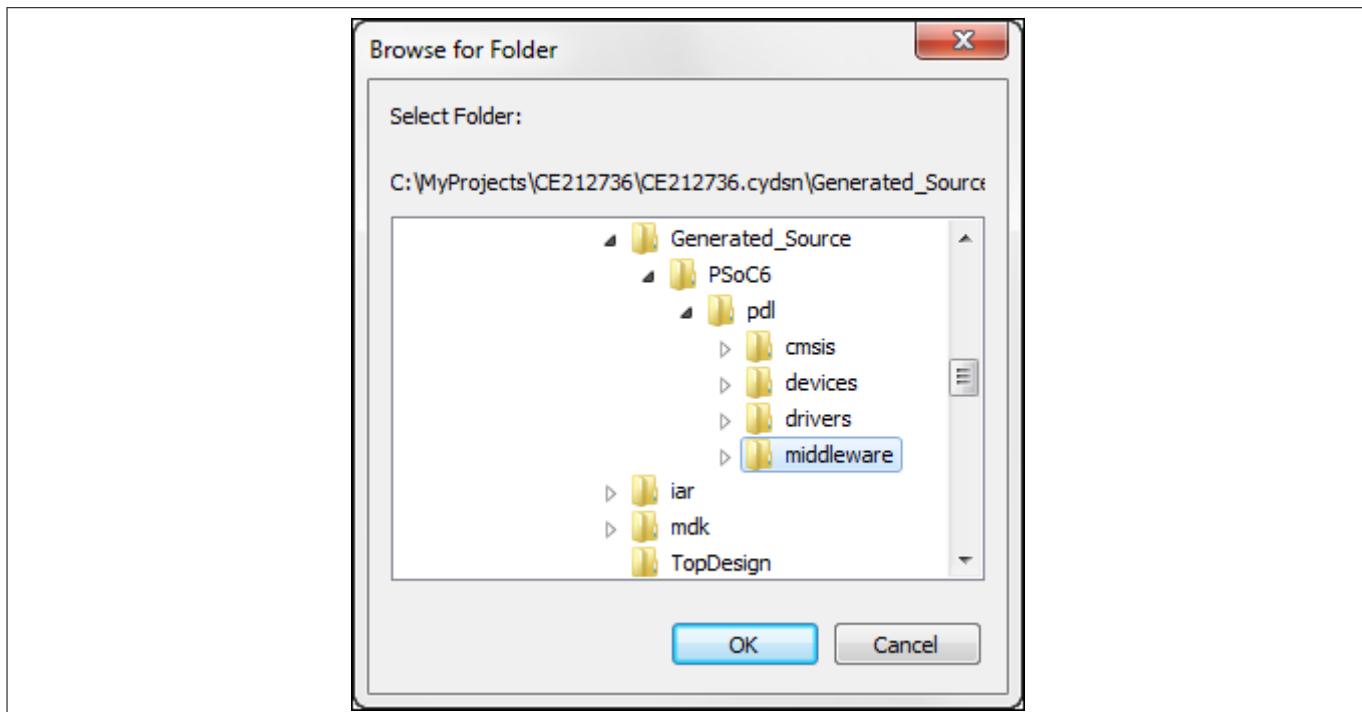


Figure 540 Setting the path to the Bluetooth® Low Energy stack

~~DRAFT~~ 5 PSoC™ 6 application notes

5.14.5.4.5 Set other required paths

There are other header files that must be included for a successful build. PSoC™ Creator provides copies of these header files in the cydsn/Generated_Source/PSoC6/pd1 folder:

- CMSIS header files: cydsn/Generated_Source/PSoC6/pd1/cmsis/include
- The ip header files: cydsn/Generated_Source/PSoC6/pd1/devices/psoc6/include/ip

Series-specific header files: cydsn/Generated_Source/PSoC6/pd1/devices/psoc6/include

Figure 541 shows these three paths set in the µVision IDE, pointing to the generated code.

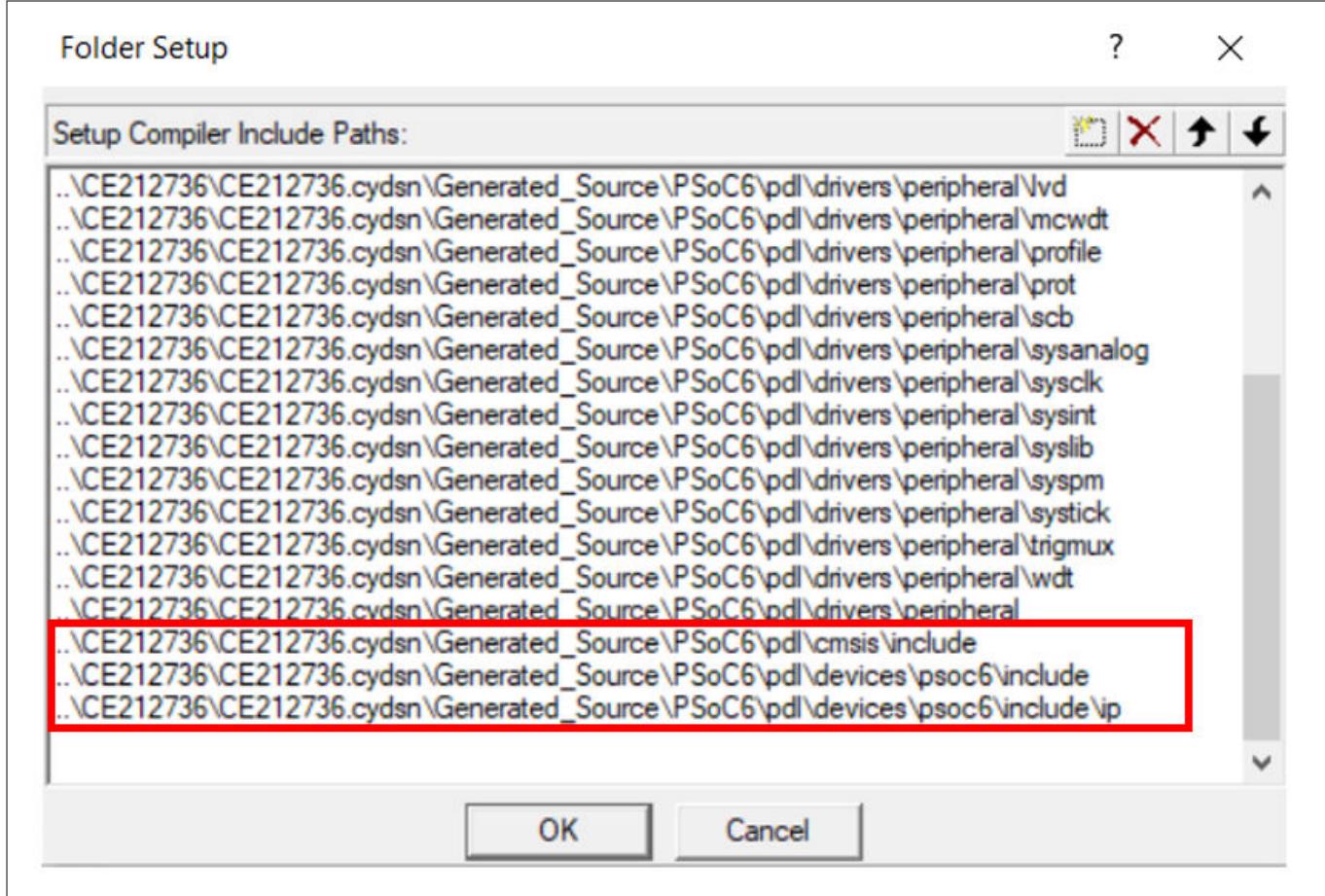


Figure 541 Paths to other required header files

Note: If you start with a PDL template project, these three paths may already be set for you. However, they point to the original files in the PDL installation, not the generated code. You should modify these paths to point to the corresponding locations in the cydsn folder. See [Where to get PDL library files](#).

5.14.5.5 Build the project in the IDE

Before compiling the code, you must configure all options required for your application to build successfully. This application note is about how to import generated code into an IDE, not about how to configure a project in any given IDE. However, [Appendix A](#) provides additional background.

With all the required files added and all paths set correctly, build your project. It should build successfully. You may encounter warnings or errors. Each IDE has a unique interface, as well as unique options, default settings, and warning and error messages. Because of the variability in IDEs, this application note cannot provide detailed guidance on handling errors. Your familiarity with your IDE will go a long way towards resolving any issues you encounter.

5 PSoC™ 6 application notes

Related to importing code, however, there are typically three kinds of errors you may encounter.

If a “file not found” error occurs for a header file, locate the actual file in your file system. Ensure that there is an include path for that file.

If a function or symbol is undefined, make sure all user, design, and library files have been added to the project, and that any header file that declares the symbol is found. For example, failing to include binary libraries causes this kind of error.

Even when all files and paths are correct, you may get a cascade of compiler or linker errors. There may be a setting in your project not configured correctly. For example, if the compiler in the IDE does not default to supporting the C99 standard, ensure that support is enabled.

5.14.5.6 Example review

This section provided an example of importing code manually into an IDE. While a single example cannot cover all situations, the tasks are straightforward:

1. Create and configure a project file in the IDE, one per CPU for a multi-CPU device. Or, start with a PDL template project
2. Add the required PSoC™ Creator generated files to the project file (for a multi-CPU device, do this for each CPU)
3. Set include paths for those files

PDL template projects provide flash configuration, linker, and startup files. If you prefer an unsupported IDE, use those as a reference. The IDE-specific files are located here: <PDL Install Folder>/devices/psoc6.

Finally, this application note teaches you the basic principles of importing code, such as what files exist, where to find them, and how to use them. You will need to apply this knowledge to your circumstances.

5 PSoC™ 6 application notes~~DRAFT~~
5.14.6 Summary

Writing software for a dual-CPU embedded system, like the PSoC™ 6 MCU, can be a daunting task. PSoC™ Creator simplifies that process. You create and configure a design using a friendly UI. PSoC™ Creator generates all the code required to implement that design, literally at the click of a button. You focus on real value, creating firmware on top of that generated code.

You can develop that firmware entirely in PSoC™ Creator, but many developers and organizations have a preferred development system.

This application note showed you how to use PSoC™ Creator generated code in a third-party IDE: either a supported IDE via export and import, or any IDE via manually importing the required files. You learned about the different kinds of generated files, where to find the files, and how to add them to the IDE's project.

The knowledge contained in this application note enables you to combine the best of both worlds: high-quality generated code to shorten development time, and your preferred IDE.

5 PSoC™ 6 application notes

References

-
- 1
- 4

Application notes

AN210781 – Getting Started with PSoC™ 6 MCU with Bluetooth® Low Energy (BLE) Connectivity	Describes PSoC™ 6 MCU with Bluetooth® Low Energy Connectivity devices and how to build your first PSoC™ Creator project
AN215656 – PSoC™ 6 MCU: Dual-Core CPU System Design	Describes the dual-core CPU architecture in PSoC™ 6 MCU, and shows how to build a simple dual-CPU design
AN225588 – PSoC™ 6 MCU Importing Generated Code into an IDE	Describes the analogous process for ModusToolbox™ software and configurators, rather than the PDL and PSoC™ Creator

PSoC™ 6 MCU

PSoC™ 6 MCU home page	Provides access to all PSoC™ 6 MCU resources
PSoC™ 6 MCU community	Discusses PSoC™ 6 MCU questions

Peripheral Driver Library (documents installed with the PDL)

Peripheral Driver Library v3.0 User Guide	Overview of the library and how to use it
Peripheral Driver Library API Reference	Detailed technical reference for the API

PSoC™ Creator

PSoC™ Creator Product Page	Access to downloads, training, components, and more
PSoC™ Creator Quick Start Guide	Get up and running quickly
PSoC™ Creator User Guide	Comprehensive manual

Development kit documentation

[CY8CKIT-062-BLE PSoC™ 6 BLE Pioneer Kit](#)

~~5 PSoC™ 6 application notes~~

~~DRAFT~~ A Appendix A - configuring an IDE project file

For the code to build successfully, you must configure the IDE project file with settings for various options. Because each IDE has a unique UI and default settings, how to configure an empty project for any particular IDE is beyond the scope of this application note. However, a general concept of what you need to configure is invaluable. Armed with that knowledge, you can locate and set the appropriate options in the IDE.

[Table 117](#) lists several options required to configure an IDE project file. For any given IDE, there may be additional options. The default setting for any option may already be set correctly.

Table 117 IDE options

Type	Option	Notes
Device	Target	If the IDE supports a particular device, select the device. This typically sets other device-dependent options automatically. If the device is not listed in the IDE, choose a generic Cortex® M4 or M0+ device, and ensure that all device-dependent options are set correctly.
Compiler	C99 Support	Enable support for the C99 standard.
	Floating Point	Enable floating point support for the CM4 CPU.
	Optimizations	Set the compiler optimization level appropriate for your build.
	Debugging	Generate debug information for a debug build.
	Include Paths	Set include paths. This application note discusses all paths related to generated code.
	Debug Symbol	Conditionally-compiled PDL code requires a debug symbol be defined. For a debug build, define DEBUG. For a non-debug build, define NDEBUG.
Linker	Device Symbol	Conditionally-compiled device-specific PDL code requires the correct symbol be defined. This symbol controls which device-specific header file is used for the build. See <code>cy_device_headers.h</code> .
	Command File	Specify the path to the linker script. The PDL provides linker files for supported IDEs. If the IDE has device-specific support, this may be set automatically.
Debugger	Other Settings	Set other linker options for your build. For example, generate a linker map file, output debug information, generate a log file, and so forth.
	Debug Connection	Specify a supported debug connection (probe), such as J-Link or CMSIS-DAP.
	Connection Settings	Based on your debug connection, specify various connection settings such as reset options, connection speed, cache options, download options, and so forth.
	Memory Configuration	Specify memory regions for the device. If the IDE has device-specific support, this may be set automatically.
	Register Description	The PDL provides a system view description (SVD) file for register-level debug information. If the IDE has device-specific support, this may be set automatically.
	Flashloader	Specify the flashloader to use for downloading the executable to the device. If the IDE has device-specific support, this may be set automatically.

5 PSoC™ 6 application notes**B Appendix B - using generated code for learning**

Even if you cannot import PSoC™ Creator generated code because of your circumstances, that code is still a valuable resource to assist firmware development.

In this case, the principal value of the PSoC™ Creator generated code is as a learning resource for how to use the PDL. There are several areas in which this is a significant help, including but not limited to:

- Clock configuration – see system_psoc6_cm0plus.c, system_psoc6_cm4.c, and cyfitter_cfg.c
- Interrupt configuration – see cyfitter_sysint_cfg.c
- Pin configuration – see cyfitter_gpio.h and cyfitter_cfg.c
- Peripheral configuration – see the Component-specific .c and .h files
- PDL function API – see the Component-specific .c and .h files

In each case, you can extract code snippets, particular functions, algorithms, or even complete files from within the generated code and use them in your own code. For example, you may choose to copy the configuration structures for a peripheral or refer to these structures as you write your own code.

PSoC™ Creator generates a Component-specific API on top of the PDL API. The Component API typically has a function defined that maps 1:1 to the PDL API. The Component API provides required hardware parameters based on the design. It also typically includes Start() and Stop() functions that make PDL API calls (in the correct sequence) required to initialize, enable, or terminate a particular peripheral. You can explore the Component API to see how it uses the PDL API.

In addition, because most of the PDL is provided as source code, you can explore the PDL source files to see what registers are used to control features and behavior.

There are dependencies among the various generated code files. The code generation process defines symbols and uses its own naming conventions. It creates a complete API for each Component. You decide how much to use directly, and how much to adapt to fit your firmware development processes.

~~5 PSoC™ 6 application notes~~

~~DRAFT~~ 5.14.9 Revision history

Document version	Date of release	Description of changes
*A	2017-07-26	First public release.
*B	2018-09-01	Updated to new AN template Updated for PSoC™ Creator UI change for IDE-specific files Updated to allow for single-CPU devices
*C	2019-04-16	Added discussion of ModusToolbox™ software and analogous process Added discussion of tool compatibility with PSoC™ Creator and PSoC™ Programmer Update to latest AN template
*D	2020-06-05	Updated ModusToolbox™ information for v2.x, with links to collateral
*E	2021-03-08	Updated to Infineon Template
*F	2022-07-21	Template update

5.15 AN221111 PSoC™ 6 MCU designing a custom secured system

About this document

•
1
5

Scope and purpose

This document teaches you what is required to create a secured system, from the boot process all the way to your application execution with PSoC™ 62/63 devices. The companion code example CE234992 PSoC™ 6 MCU: Security Application Template implements many of the topics described in this document.

Intended audience

This application note is intended for developers that want to learn more about the security features of the PSoC™ 62/63 family of devices.

More code examples? We heard you.

To access an ever-growing list of hundreds of PSoC™ code examples, please visit our [code examples web page](#). You can also explore the video training library [here](#).

5 PSoC™ 6 application notes

~~DRAFT~~ 5.15.1 Introduction

This application note teaches you about the security features of the Infineon PSoC™ 62/63 families and how to utilize these features to secure your application. Knowing this information will allow you to design a secure system that fits the needs of your application.

Note: This application note does not include the PSoC™ 64 Secured MCU, because it does not allow the flexibility described in this document.

This is an advanced application note and assumes that you are familiar with the basic PSoC™ 6 MCU architecture described in the device datasheet and the technical reference manual (TRM).

This document will cover the following topics:

- Assessing your security needs
- Security Features of the PSoC™ 62/63
- Understanding the PSoC™ 6 bootup sequence
- What is a Chain of Trust (CoT) and do you need it?
- What is and how to use a public/private key pair
- Signing your application

This application note does not cover side channel attacks where an attacker tries to gain information from a microcontroller by exploiting weaknesses in the device implementation, which includes timing, power-monitoring, electromagnetic attacks, and micro probing.

Use the code example [CE234992 “PSoC6™ MCU: Security Application Template”](#) as a companion document with this application note. Most code snippets in the application note are copied directly from CE234992. Also, CE234992 can be used as a template for a secured application. It supports the following features:

- Secure boot
- Bootloader (MCUboot)
- Dual CPU (CM0+ and CM4) support
- Real time RTOS (FreeRTOS)
- Device firmware update (DFU)
- 1 M, 2 M, 512 K and 256 K PSoC™ 61/62/63 devices

There are two generations of PSoC™ 6 devices and they have some minor differences in the way they operate or how they are configured. The table below identifies the PSoC™ 6 families and which generation they belong. This document will refer to these parts as 1st and 2nd Generation PSoC™ 6 devices.

Table 118 PSoC™ 6 device generations

1 st Generation PSoC™ devices	CY8C61x6, CY8C62x6, CY8C63x6 CY8C61x7, CY8C62x7, CY8C63x7
2 nd Generation PSoC™ devices	CY8C61x4, CY8C62x4 CY8C61x5, CY8C62x5 CY8C61x8, CY8C62x8 CY8C61xA, CY8C62xA

5 PSoC™ 6 application notes

5.15.2 System security

Twenty-five years ago, few embedded system developers thought too much about security. Security meant that you did not release your source code to the public and you knew that few people would attempt to reverse engineer the binary code that was in your device. Also, the level of connectivity was almost non-existent, compared to today with the Internet of Things. Even the smallest electronic devices are wirelessly connected with either Wi-Fi or Bluetooth® LE. The other change is that most of the embedded devices on the market contain one of more Arm® processors and share a common SWD (Serial Wire Debugger) which makes it easy for a hacker to switch from one device to another without much of a learning curve.

At the start of any new project, security should be discussed and determine what needs to be protected. Usually the first thought is to protect the code and hardware from being hacked, but you should also consider any user data that may be collected or transmitted to the cloud or another location. It is not just that you are protecting code from being read but, protecting it from being modified or misused as well. The same goes for user data. Manipulating user data can be just as dangerous as reading the data. When evaluating what should be protected in your system, the list below is some of the items that should be considered. Not all the items in this list are required for every application, but you need to decide what is important.

- Protect OEM IP (code/data/keys)
- Protect end user data and keys
- Confirm code integrity (CRC or hash)
- Authenticate code origin (signing)
- Authenticate firmware updates
- Protect hardware from unauthorized usage
- Isolate the two CPU's memory space (dual core)
- Protect data transmitted to and from the device

Protecting your firmware IP is probably the most obvious concern. If someone were to download and reverse engineer your code, they could more quickly get to market and pose direct competition. Although this is a valid threat, a security breach into your system could prove much more devastating to your product line than just added competition. If it was found that your device could be compromised, user data stolen, or taken control of, it could end the sales of the product overnight.

Most of the attacks usually come from outside the microcontroller, but there are attacks that can come from inside as well. Some attacks are accidental, for example, one CPU might crash due to a coding error and write data over the second CPU's stack or data area. Although this is not a malicious attack, the outcome can be just as serious.

When using two cores and code from multiple sources, without any protection applied, both CPUs have full access to the entire memory space. By isolating and protecting these CPUs from each other, you can guarantee that malicious or accidental failure does not cause system instability, failure, or misuse.

There are four main ways products can be hacked:

- **Direct access to the debug port.** With the use of common debug tools and dongles, accessing or reprogramming firmware or examining internal data is easy. Most common CPUs are based on just a few architectures, so hacking or reverse engineering a product is easy if the device is left unsecured.
- **Direct connection to a communication port** such as SPI, I²C, or a UART. Depending on the firmware, this connection may allow firmware to be read or updated with non-sanctioned software. If one person decodes the protocol and posts it on the internet, you can have many hackers gaining access.
- **Network connections** such as Bluetooth®, Wi-Fi or ethernet. This has become the standard method of hacking because it does not require physical contact. The perpetrator can be out on the street or half way around the world via the internet. The firmware stacks to handle these interfaces are complex and difficult to fully test against all attacks.
- **Third-party code** that's installed in the device after it has been shipped. For example, the applications that you download into your smart phone. Malicious code in third-party applications can be dangerous because

~~5 PSoC™ 6 application notes~~

~~DRAFT~~
the code is already inside the device. This code must be kept inside a controlled environment with limited access to memory and system utilities.

Not all applications require the full gamut of security tools, you may not need to read all sections of this application note. [Table 119](#) provides a few application examples and which sections of the [Security features](#) chapter you should consider reading.

Table 119 Reference sections for security tools

Example	Firmware updates	Intellectual property secure	Sections to read (Chapter 3)	Comment
Simple toy #1	No	No	NA	Device does not need to be secure.
Simple toy #2	No	Yes	<ul style="list-style-type: none"> • Device lifecycle • Debug ports 	Device does not communicate with other devices or connect to the cloud. Disabling the debug ports is all that is required.
Simple appliance	Yes	Yes	<ul style="list-style-type: none"> • Device lifecycle • Debug ports • Authenticate • CoT 	Device is not connected to wireless devices. Code is updated with a dedicated link or device. Code requires authentication.
Simple connected appliance	Yes	Yes	<ul style="list-style-type: none"> • Device lifecycle • Debug ports • Authenticate • CoT 	This device may be connected wirelessly to cloud or other devices. Code updates need to be authenticated.
Wi-Fi/Bluetooth® LE connected device	Yes	Yes	<ul style="list-style-type: none"> • Device lifecycle • Debug ports • Authenticate • CoT • Protection units 	Device is connected with a complex stack (Bluetooth® LE, Wi-Fi, etc.). The stacks should be protected/isolated. Also like the previous example, code should be authenticated.

5 PSoC™ 6 application notes

5.15.2.1 Security basics

A security plan is how you define access to your device. You can split this into two types, external and internal.

5.15.2.1.1 External access

By default, external access is only through the debug port unless the designer adds an additional communication port such as UART, I2C, SPI, Wi-Fi, Bluetooth® LE, and so on, which allows access to the memory. A bootloader can be used to update the firmware with your application using a communication interface of your choice. Unlike the debug ports, the developer can limit access to internal memory from the outside. It allows an alternate path to update the secured code and data but can be just as dangerous as the debug port if written incorrectly.

It is common and recommended that all debug ports are disabled, and firmware is updated only with a bootloader. How the data is transmitted to the device and whether it is encrypted or not is up to the designer. Listed below are four different secure system strategies for the external part of your security plan. Each of the pieces to implement these strategies are discussed in the [Security features](#) chapter.

- **Firmware updates with a hardware debugger:** This is usually not thought of as a secure system, but if the hardware is installed such that a third party cannot get direct access, then it may be secure. Flash areas can be blocked from writing so that any internal hack could not change or replace the application. The device can be put in a SECURE lifecycle stage with the debug port open, which will force the firmware to be authenticated with a public key each time the device comes out of reset.
- **No access to debug port; bootloader for updates:** The debug access ports are disabled so that after the initial programming, the only way to update the firmware is to provide a bootloader. The security level of the bootloader is determined by the implementation. The PSoC™ 6 family includes a Crypto block that can be used to generate the hash, encrypt, or decrypt to implement secured bootloader features. For systems that do not include an internal crypto block like in PSoC™ 6, [Infineon OPTIGA™ embedded security](#) products can provide a good alternative.
- **Lock down firmware; no updates:** Debug access ports are disabled and there is no provision to allow for bootloading. This may be the most secure, but there is no way to perform bug fixes or add future enhancements.
- **Hybrid:** A custom secure design that allows partial access to the memory via the debug port. This may fit your custom needs but will require more work and thought.

5.15.2.1.2 Internal access

The internal portion of your security plan is how the code and data are protected inside the device. Each application will be unique depending on how your CPUs are utilized, what the memory requirements are, and what is running on each CPU.

- **No protection:** No protection or limits on the CPUs to access all data and code on the device. Because the code is all from the developer, it is assumed there is no malicious code. There also is no protection from one CPU firmware bug possibly corrupting the SRAM used by the other CPU.
- **Isolated CPUs:** In this model, the two CPUs are totally isolated with perhaps one area of shared memory. Protection units are used to disallow any accidental reading/writing of data between CPUs. Communication between CPUs is achieved with shared memory or IPC hardware.
- **Secured CPU:** This model uses the CM0+ CPU as the secured processor. In addition to being used for system calls, CM0+ will be used for all secure data and secure functions. Protection units are configured and owned by CM0+, so CM4 will not have access to secure data/code unless required by the system design. This is by far the most secured approach.

~~5 PSoC™ 6 application notes~~

~~DRAFT~~ 5.15.2.2 Basic definitions

Before continuing, you must understand some terms that will be used throughout this document. Many of these terms will be discussed in more detail in the following sections of this application note.

Table 120 Abbreviations and definitions

Chain of Trust (CoT)	Chain of Trust is established by validating the blocks of software starting from the root of trust located in the ROM. The root of trust begins with the Infineon code residing in the ROM that cannot be altered.
Code signing	Process of calculating a hash of the code binary and encrypting the hash with a private key and appending this to the code binary.
Dead access restrictions (DAR)	Determines what resources are accessible via the debug port when in DEAD mode. DEAD mode occurs if an error is found during the boot sequence. The DAR are stored in eFuse.
Debug access port (DAP)	Interface between an external debugger/programmer and PSoC™ 6 MCU for programming and debugging. This allows connection to one of three access ports (AP), CM0_AP, CM4_AP, and System_AP (Sys_AP). The System_AP can access only the SRAM, flash, and MMIOs, not the CPU.
Digest	The output of a cryptographic hash function is often called a message digest or digest. This digest is then encrypted with a private key to form a digital signature.
Digital signature	Encrypting of the digest (hash of a data set). For example, the encrypted hash of the user application.
Elliptic-curve cryptography (ECC)	ECC is an asymmetric encryption system that uses two keys. One key is private and should not be shared, and the other is public and can be read without loss of security. ECC is a more modern method than RSA and requires a smaller key than RSA for the same level of security.
eFuse	One-time programmable (OTP) memory that by default is 0 and can be changed only from 0 to 1. eFuse bits may be programmed individually and cannot be erased.
Factory hash	Calculated hash (SHA-256) of the system trim values and Flash boot. This hash is truncated to 128-bit (MSbs) and stored in the eFuse prior to leaving Infineon. This is used to validate that trim values and Flash boot (part of the boot sequence) have not been compromised.
Flash boot	Part of the boot system that performs two basic tasks: <ol style="list-style-type: none"> 1. Sets up the debug port based on the lifecycle stage. 2. Validates the user application before executing it. <p>Flash boot executes after ROM boot.</p>
Flash (User)	Flash memory that is used to store your application code. It is non-volatile by may be reprogrammed.
Hash	A crypto algorithm that generates a repeatable but unique signature for a given block of data. This function is non-reversible.
IP	Intellectual property. This can be both code and data stored in a device.
Inter-process communication (IPC)	Inter-processor communication. Hardware used to facilitate communication between the two CPU cores.

(table continues...)

5 PSoC™ 6 application notes

Table 120 (continued) Abbreviations and definitions

Lifecycle stage (LCS)	The LCS is the security mode in which the device is operating. To the user, it has only four stages of interest: NORMAL, SECURE, SECURE_WITH_DEBUG, and RMA.
Memory protection unit (MPU)	The MPU is used to isolate memory sections from different bus masters.
MMIO	Memory-mapped input/output, usually refers to registers that control the hardware I/O.
Normal Access Restrictions (NAR)	Normal access restrictions determine what memory resources are accessible via the debug port when in NORMAL mode. The NAR is stored in SFlash.
Protection context (PC)	The PC allows each bus master a security state level from 0 to 15. A bus master can be assigned a PC value that stays static or that is changed during application execution. PC provides a more precise way of applying memory restrictions. PC=0 is a special case which allows any bus master to have full access to the entire memory space including registers. The PC state works together with protection units.
Protection units	Hardware blocks that are used to limit the bus master access to the memory (SRAM, ROM, flash) or hardware (peripheral) registers. They include MPU, PPU, and shared memory protection unit (SMPU).
Peripheral protection unit (PPU)	PPUs are used to restrict access to a peripheral or set of peripherals to only one or a specific set of bus masters.
Protection state	Three possible states: NORMAL, SECURE, and DEAD. Each state may be configured by the user. The NORMAL protection state configuration is stored in SFlash, but SECURE and DEAD state configurations are stored in the one-time programmable eFuse.
Public-key cryptography (PKC)	Also known as asymmetrical cryptography. Public-key cryptography is an encryption technique that uses a paired public and private key (or asymmetric key) algorithm for secure data. It is used to secure a message or block of data. The private key is used to encrypt data and must be kept secured, and the public key is used to decrypt but can be disseminated widely.
Public key	When using asymmetrical cryptography such as RSA or ECC, a public key is used to validate firmware that was signed by the private key. It can be shared, but it should be authenticated or secured so it cannot be modified.
Private key	When using asymmetrical cryptography such as RSA or ECC, the private key is used to sign (encrypt the hash) of firmware after it is built but prior to being loaded into the device. It must be kept in a secure location, so it cannot be viewed or stolen.
RMA	Return Merchandise Authorization
ROM	Read-Only Memory is non-volatile and is programmed as part of the fabrication process and cannot be reprogrammed.
ROM Boot	After a reset, the CM0+ starts executing code that has been programmed into ROM. This code cannot be altered.

(table continues...)

5 PSoC™ 6 application notes

Table 120 (continued) Abbreviations and definitions

RSA-nnnn	An asymmetric encryption system that uses two keys. One key is private and should not be shared and the other is public and can be read without loss of security. The encryption/decryption is controlled by a key that is commonly 1024, 2048, or 4096 bits in length (RSA-1024, RSA-2048, or RSA-4096).
Secure Access Restrictions (SAR)	Secure access restrictions determine what memory resources are accessible via the debug port when in SECURE mode. The SAR is stored in eFuse.
Secure hash	Calculated hash (SHA-256) of the system trim values, Flash boot, TOCs, and user public key. This hash is generated during the transition to SECURE and stored in eFuse. This insures that the OEM's public key has not been corrupted maliciously or accidentally.
Security plan	The security plan is the set of rules that the designer imposes to determine what resources are protected from outside tampering or between the internal CPUs.
Serial memory interface (SMIF)	A SPI (serial peripheral interface) communication interface to serial memory devices, including NOR flash, SRAM, and non-volatile SRAM.
Supervisory Flash (SFlash)	Supervisor flash memory. This memory partition in flash contains several areas that include system trim values, Flash boot executable code, public key storage, etc. After the device transitions into a SECURE mode, it can no longer be modified.
SHA-256	SHA-256 is a common cryptographic hash algorithm used to create a signature for a block of data or code. This hash algorithm produces a 256-bit unique signature of the data no matter the size of the data block.
Shared memory protection unit (SMPU)	SMPUs are used to allow access to a specific memory space (flash, SRAM, or registers) to only one or a specific set of bus masters.
System calls	System calls are functions such as flash write commands that are executed by the Arm® Cortex® M0+ CPU (CM0+) from ROM. These system calls may be called from either the CM4 or CM0+ CPUs, or via the debug ports.
Table of Contents1	(TOC1) An area in SFlash that is used to store pointers to the trim values, Flash boot entry points, etc. It is used only by the boot code in the ROM and is not editable by the designer.
Table of Contents2	(TOC2) An area in SFlash of the PSoC™ 6 MCU that is used to store parameters and pointers to objects used for secure boot. Locations of two application pointers (Application1 and Application2) are stored here, but the second one is optional. The first pointer, Application1, must point to the first executable user code, which may be the bootloader or just the application. The table of contents also contains some boot parameters that are settable by the system designer. A duplicate of TOC2 is written in the adjacent page of flash for redundancy. This duplicate is called "RTOC2". If TOC2 is found to be invalid for any reason, RTOC2 is evaluated. Both these structures are protected with a CRC, and are part of the Secure Hash. Definition of the TOC2 structure can be found in section 4.1 .

~~5 PSoC™ 6 application notes~~

~~5.15.3 Security features~~

PSoC™ 6 MCUs has several security features that are used to build a custom secured system. In this chapter, each of these components will be described. The combination of these features can provide a secured system that meets most security requirements.

- eFuse
- Device lifecycle
- Secured boot sequence
- Chain of Trust (CoT)
- Code signing
- Protection units and protection context
- Debug port configuration

The PSoC™ 62/63 family is a dual-core device with a CM0+ and a CM4 CPU. The CM0+ is usually considered the secured processor and is the one that performs all system calls and runs the secured boot sequence. The “main()” function in the user project enables CM4 only after CM0+ has run the boot process. The CM4 is usually considered the application processor because it can run faster and is more powerful than CM0+. The CM0+ is the logical choice to configure your secure elements, because it runs before CM4. This allows your application to have your secure configuration complete before enabling CM4.

It is also important to understand that in all but the simplest secure system, all these security features work together:

- **eFuse block:** System hash values and security attributes are stored in the immutable efuse block.
- **Device lifecycle stage (LCS):** The LCS dictates how the device boots up and which security attributes to enable, such as the SAR, NAR and DAR.
- **Chain of Trust:** This dictates that the validation of the user application code is linked all the way back to the device ROM. The code verification is accomplished using the user’s public key stored in SFlash. A hash is calculated for the Sflash area and stored in eFuse. Any changes in the eFuse, public key, or user code will be detected and boot will fail.
- **Protection units:** These protect or isolate the code and data from different bus masters. They are also used to secure the hardware such as flash and effuse programming. The user can also protect other hardware such as GPIOs or communications ports.
- **Debug port:** The debug port is initially used for programming and debug of the user application, but should be disabled after the device is in SECURE lifecycle stage.

5.15.3.1 eFuse

eFuses are simple devices but an integral part of security for the PSoC™ 6 devices. There are 1024 eFuse bits which default to “0” and can only be programmed to a “1”. Once programmed, they cannot be erased back to “0” ever, not even by Infineon.

Note: When programming eFuse, the V_{DDIO0} pin must be connected to a 2.5-volt supply.

eFuse bits are stored in the programming file (intel hex) using the address range 0x9070_0000 to 0x9070_03FF. This range is virtual and cannot be used for direct read or write operations. The eFuse library uses the offset value to read 8 bits of eFuse data at a time.

[Table 121](#) identifies the usage areas of interest in the eFuse block including the user area. Note that there is some difference in the eFuse usage between 1st generation and 2nd generation devices. [Table 118](#) defines which family of parts belong to the 1st and 2nd generations devices

5 PSoC™ 6 application notes

Table 121 eFuse data

Data	eFuse bit address in programming file	eFuse block byte offset (read)	Size	Notes
Secure hash	0x9070_00A0	0x14	16 bytes (128 fuses)	128 bits of the MSbs of a 256-bit hash. Created when moving to SECURE mode.
Secure hash zeros	0x9070_0130	0x26	1 byte (8 fuses)	Number of zeros in the secure hash. This value is to ensure that the hash cannot be altered without being detected.
DAR	0x9070_0138	0x27	2 bytes (16 fuses)	Dead access restrictions
SAR	0x9070_0148	0x29	2 bytes (16 fuses)	Secure access restrictions
Lifecycle	0x9070_0158	0x2B	1 Byte (8 fuses)	Lifecycle status
Factory hash	0x9070_0159	0x2C	16 bytes (128 fuses)	128 bits of the MSbs of the 256-bit hash. This is generated and stored before the device leaves the factory.
Factory hash zeros	0x9070_01E0	0x3C	1 byte (8 fuses)	Number of zeros in the factory hash. Ensures that the hash cannot be altered without being detected.

1st generation parts

User data	0x9070_0200	0x40	64 bytes (512 fuses)	User eFuse area
-----------	-------------	------	----------------------	-----------------

2nd generation parts

Asset hash	0x9070_0200	0x40	16 bytes (128 fuses)	Similar to the factory hash, but it does not include trim values so it will be the same for all parts with the same silicon and firmware version. (This is generated and written to eFuse automatically)
Asset hash zeros	0x9070_0280	0x50	1 byte (8 fuses)	Number of zeros in the asset hash. Ensures that the hash cannot be altered without being detected. (This is calculated and written automatically)
User data	0x9070_0281	0x51	47 bytes (376 fuses)	User eFuse area

~~5 PSoC™ 6 application notes~~

~~5.15.3.2~~ Device lifecycle

The device lifecycle is a key aspect of the PSoC™ 6 MCU's security. Lifecycle stages follow a strict irreversible progression dictated by programming eFuses bits (changing a fuse's value from '0' to '1'). This system is used to protect the internal device data and code at the level required by the customer. Lifecycle stages are governed by the LIFECYCLE_STAGE eFuses and can only be advanced to the next lifecycle state as shown in [Figure 542](#). For example, once in the SECURE lifecycle stage, the device can never return to the NORMAL or VIRGIN state.

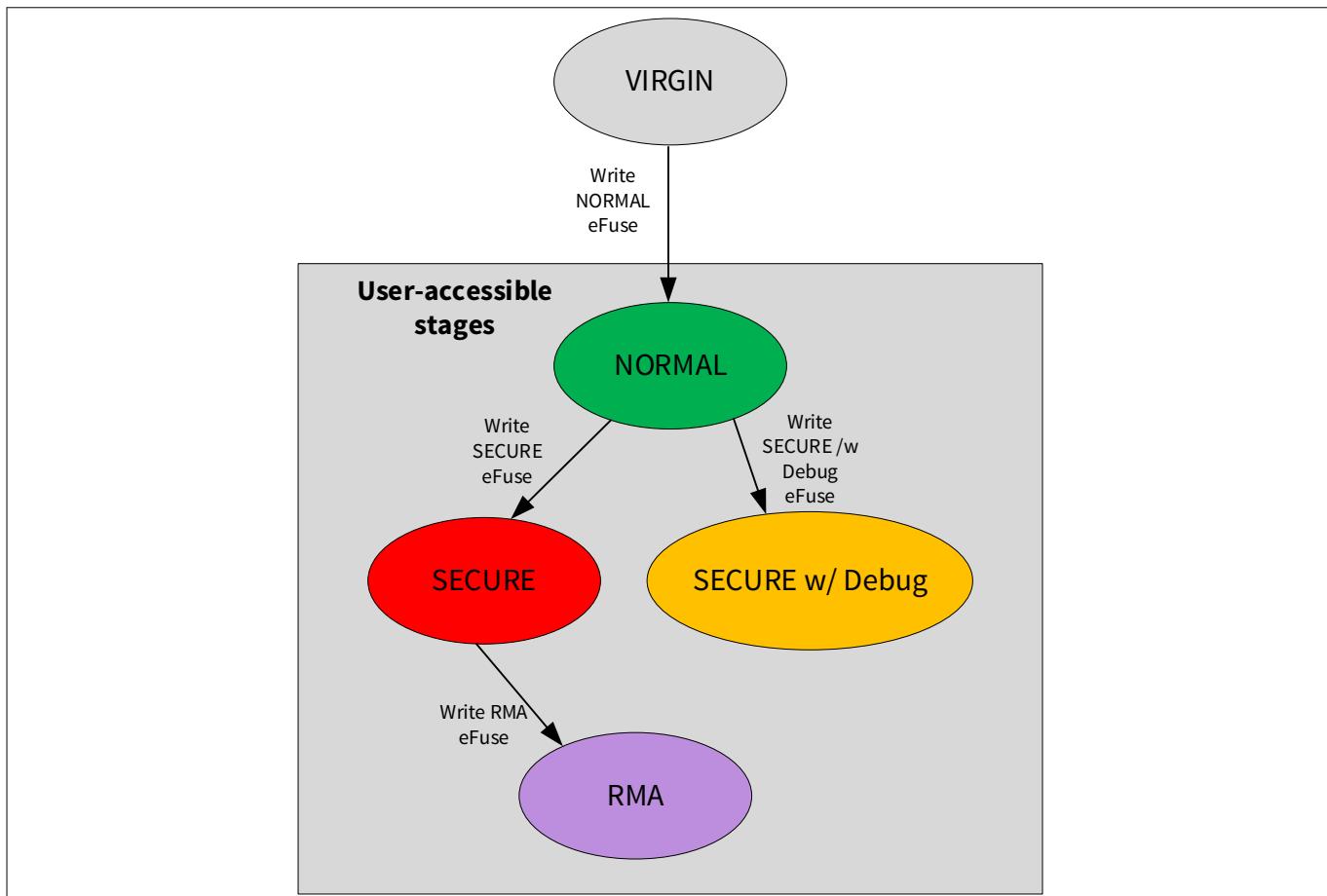


Figure 542 **Device lifecycle**

5.15.3.2.1 **VIRGIN**

This is the initial lifecycle stage of the device when manufactured. During this stage, trim values and Flash boot are written into SFlash. Parts in this stage never leave the factory. Once all factory tests are passed and the factory hash is written, the device will be transitioned to the NORMAL lifecycle stage.

5.15.3.2.2 **NORMAL**

This is the lifecycle stage in which the parts are sent to customers. By default, users have full debug access and may program all user flash including certain areas in SFlash such as user SFlash, TOC2, and public key. To allow the OEM to check the data integrity of trims, Flash boot, and other objects from the factory, a hash (SHA-256 truncated to 128 bits) of these objects is stored in eFuse, called the "factory hash". The factory hash is not verified during boot, but should be verified before moving to the SECURE lifecycle stage.

5 PSoC™ 6 application notes

Normal access restrictions (NAR)

Access restrictions to the debug ports may be modified in the NORMAL lifecycle state. Normal access restrictions (NAR) are located in SFlash (0x1600_1A00) and use the same format as the SARs in eFuse. (See section [Appendix C - Debug port access settings](#) for NAR, SAR, and DAR formats) Once set, they can be changed to be more restrictive, but not less, by the flash write system call.

Note that the NAR settings are not considered secure because a user could alter the NAR by writing a custom flash write function instead of using the flash write system call. The system calls that write the flash memory will not allow erasing the bits in the NAR SFlash area.

Using NAR to secure a device

If you are considering using NAR to secure a device, you should set up the protection context registers and move all bus masters (CM0+ and CM4) to a protection context other than 0. This will disallow any direct access to the flash programming registers, other than the system call infrastructure that will not allow reducing the security of the normal access restrictions. This should be done preferably by CM0+ at the beginning of the CM0+ application (or bootloader) before running any user code in CM0+ or enabling CM4. This will effectively lock out any access to the registers required to program internal flash memory and force you to use flash write system calls to the program flash.

Note: *Using the system calls for programming of the NAR values will only allow you to increase the access restrictions, not reduce them.*

If you set the NAR to disable the debug ports and change the protection context to $PC \neq 0$, you can protect access to your internal code and block anyone from accessing the code or debugging. This can be an attractive option for some applications. If you plan on upgrading your code in the future and have disabled the debug port, you will need a bootloader and a way to transfer the new code to the device other than the debug port before setting the NAR bits. One advantage of using NAR to lock out debug is that you can easily program the bits during runtime and it does not require the 2.5 volts connected to the VDDIO0 pin. One reason for programming the NAR values during runtime is that you could use your project application to disable the debug ports right before shipping the product after the device has been fully tested.

By default, in the NORMAL lifecycle stage, the user application code is not validated. There is an option to force your code to be validated with the public key while in NORMAL mode. This is a good way to validate that you have everything set up correctly before you advance to SECURE mode, because it boots the same way. If you move to the SECURE lifecycle stage and do not have the public key stored correctly or did not write the TOC2 properly, you can lock yourself out of the part and essentially “brick” the device so it is not usable, and there is no way to fix it. By using the NORMAL validation feature, you can continue to make changes until you have everything set up properly, and then you can feel more confident about advancing to the SECURE lifecycle stage. To force validation in the NORMAL LCS, you must populate the TOC2 structure and add the RSA public key, just as you would in the SECURE LCS. More details of the TOC2 structure are defined later in the application note.

Note: *In the NORMAL lifecycle stage, even if you sign your application, your system will not be secure. This is because the secure hash has not been generated and therefore, the NORMAL lifecycle state cannot verify the public key. See the [Code signing and verification](#) section for more information.*

~~5 PSoC™ 6 application notes~~~~5.15.3.2.3 SECURE~~

This is the lifecycle stage of a secured device. Before the transition to the SECURE lifecycle stage, the following tasks must be completed properly. Failure to do so could leave you with an inoperable device. The first five steps may occur in any order, but step six must be the last step. More information on performing these steps is provided later in this document.

1. Use the factory hash to verify that the device has not been tampered with since it left Infineon. The device programmer performs this check during the transition to the SECURE lifecycle stage. (See #6 below).
2. Fill in TOC2 including the CRC at the end (cymcueltool which is included in the ModusToolbox™ install is used to fill in the CRC).
3. Write the public key into SFlash (See [Appendix B - Creating crypto key pairs](#)).
4. Set the SAR and DAR in the eFuse. Once in SECURE lifecycle stage, the access restrictions cannot be altered (See section [Appendix C - Debug port access settings](#) for NAR, SAR, and DAR formats).
5. Program an application into the user flash. Depending on the SARs, a bootloader may be required to update the code in the future.
6. Transition to the SECURE lifecycle stage by setting the SECURE Lifecycle stage bit. (The CYPRESS™ Programmer tool performs eFuse programming, but the user sets the values in his code.)

In the SECURE lifecycle stage, the protection state is set to SECURE and the SAR are deployed. A secured device will boot only when the authentication of its Flash boot and application code succeeds.

After an MCU is in the SECURE lifecycle stage, there is no going back. The debug ports may be disabled depending on your preferences, which means that there is no way to reprogram or erase the device with a hardware programmer/debugger. The only way to update the firmware at this point is to provide a bootloader as part of device firmware and provide a way to invoke it.

Code should be tested in NORMAL or SECURE_WITH_DEBUG lifecycle stages before the move to the SECURE lifecycle stage. This is to prevent a configuration error that could cause the part to be no longer accessible for device programming and therefore unusable.

Note: You cannot move from SECURE_WITH_DEBUG to SECURE lifecycle stage.

5.15.3.2.4 SECURE WITH DEBUG

This is the same as the SECURE lifecycle stage, except with NAR applied to enable debugging. When in the SECURE_WITH_DEBUG lifecycle stage, the access restrictions are taken from the NAR located in SFlash. Parts put in this stage cannot be changed back to either SECURE or NORMAL stage; they are most likely discarded or destroyed after testing. Devices should not be shipped in this stage because they are not secure. It is not recommended to use this lifecycle stage during development; instead, use the NORMAL lifecycle stage with the Validate App bits set in TOC2. Only use this Lifecycle stage if you find that your device is not booting correctly in SECURE and you need to debug the problem.

~~DO NOT USE~~ 5 PSoC™ 6 application notes

5.15.3.2.5 RMA

The RMA lifecycle stage is a mechanism to allow customers to return parts that are in the SECURE lifecycle stage back to Infineon to evaluate if a defect in the device is suspected. When in RMA mode, the debug ports will be open to allow Infineon to access all hardware and memory. Before switching the device to RMA, you should erase all proprietary and sensitive data and code in the device by means such as updating the firmware (with your bootloader).

See [Appendix D - Transition to RMA](#) for more details.

5.15.3.3 Protection state

Protection state and Lifecycle stage are the same unless there is an error during the boot process or the device is in the RMA Lifecycle. If there is an error during boot, the device will be moved to the “DEAD” protection state. Access to the debug ports in the DEAD protection state is defined by the Dead Access Restrictions (DAR).

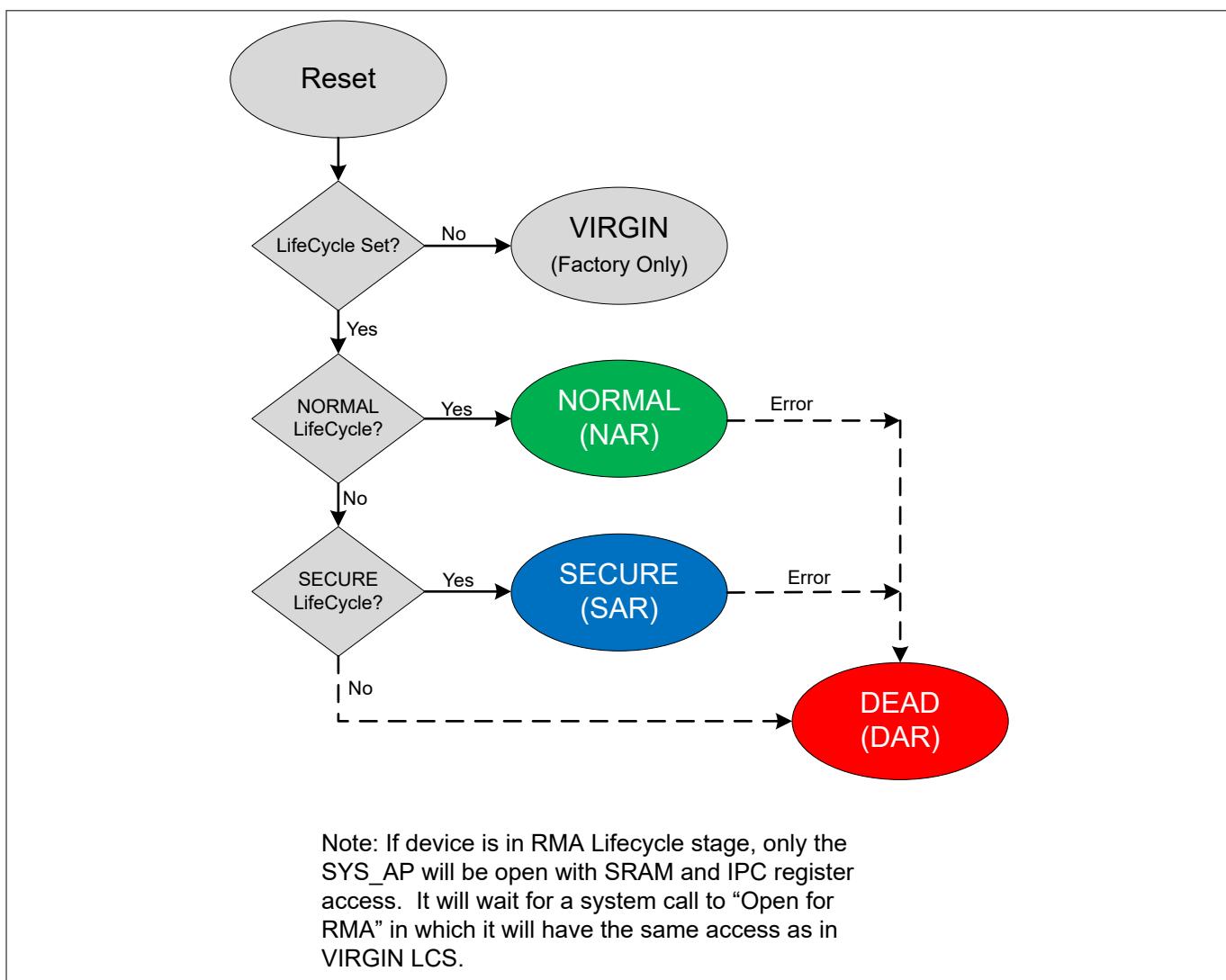


Figure 543 Protection state transitions

5.15.3.4 CM0+ boot sequence

Only the CM0+ CPU is started after a reset. It is up to the OEM’s CM0+ code to enable the CM4 after the boot sequence, if the default CM0+ startup code is not used. The default CM0+ is a binary that is provided, if the user doesn’t need to use the CM0+. It is not recommended to use for a secure system. The boot sequence differs

5 PSoC™ 6 application notes

depending on which lifecycle stage the device is in. As expected, the SECURE lifecycle stage is the only boot sequence that maintains a Chain of Trust (CoT). [Figure 544](#) shows the different paths of the four basic boot sequences.

- 1.** NORMAL (no validation, no code security)
- 2.** NORMAL with Validate (no code security)
- 3.** SECURE_WITH_DEBUG (debug only, not intended for final product)
- 4.** SECURE

DRAFT

5 PSoC™ 6 application notes

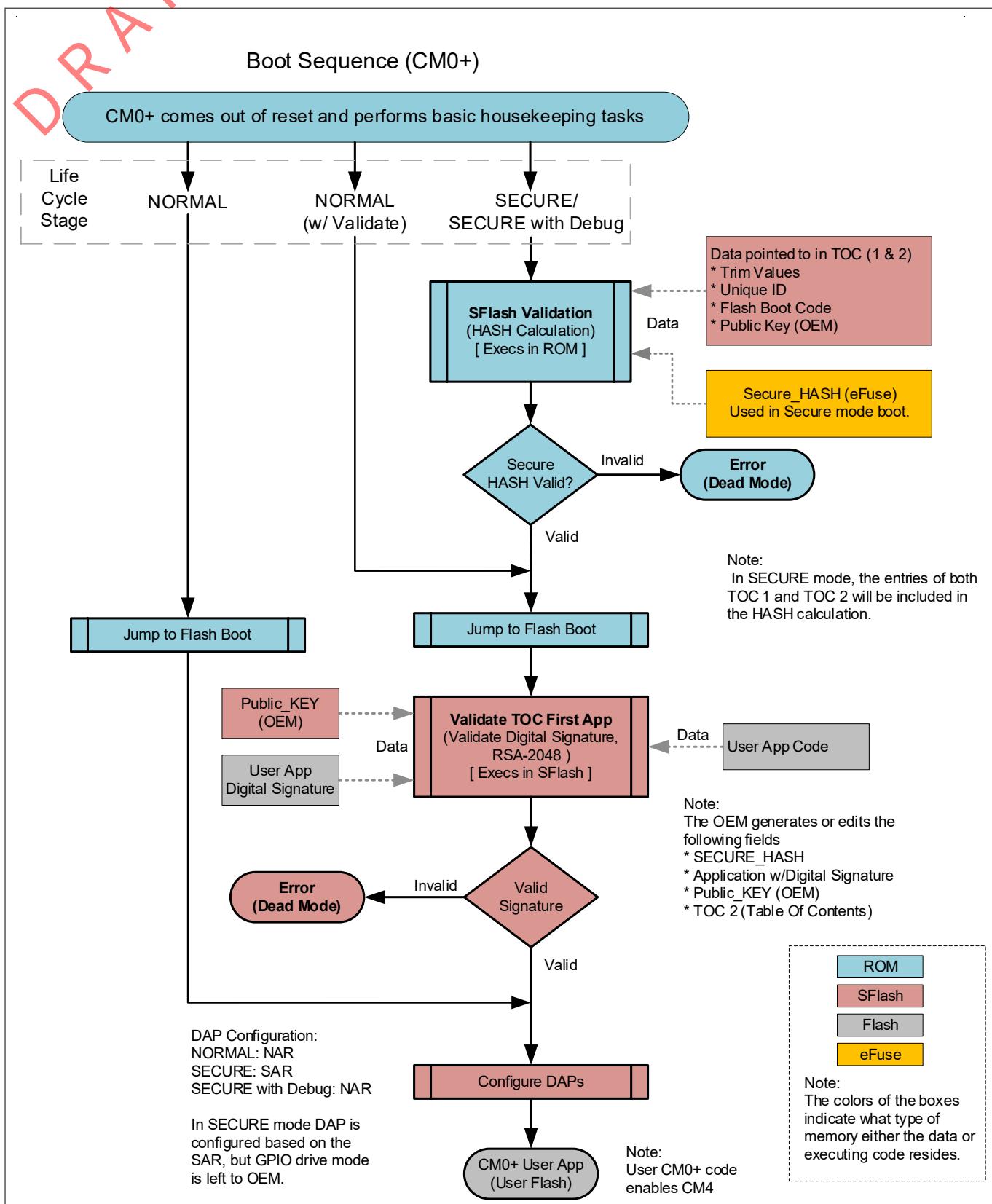


Figure 544

PSoC™ 6 CM0+ boot sequence

~~CONFIDENTIAL~~ 5 PSoC™ 6 application notes

~~CONFIDENTIAL~~ 5.15.3.4.1 NORMAL lifecycle boot

After reset and initial hardware configuration, Flash Boot (pre-programmed into SFalsh) configures the DAP port with the NARs stored in the SFlash. The GPIOs used for the debug interface are configured to interface with the debugger if any of the three debug access ports are enabled (CM0+, CM4, System). By default, it is assumed that the user code starts at 0x1000_0000 (beginning of user flash) and the first two vectors are the stack pointer and the reset vector. These values are checked to verify that they contain values that are in the available SRAM and flash respectively. If these values are outside of these memory ranges, CM0+ will jump to DEAD mode and remain in an infinite loop, and no (user) code will be executed.

5.15.3.4.2 SECURE lifecycle boot

When in the SECURE lifecycle stage, the following occur:

1. CM0+ validates the SFlash by calculating a hash of the trim values, Flash boot code, user public key, etc.
2. CM0+ compares this hash with the precalculated secure hash stored in the eFuse that was generated when the part was advanced from NORMAL to SECURE lifecycle stage by the OEM.
 - The calculated SHA-256 hash (256 bits) is truncated to its most significant 128 bits which is the same as the stored hash in the eFuse.
3. If the calculated hash matches what is stored in the eFuse, the trim (calibration) values from the SFlash are used to configure the hardware for optimal operation.
4. CM0+ executes Flash boot (in the SFlash) and validates the Table of Contents2 (TOC2). The TOC2 contains information about the location of public key, start of user code, application format, user configuration options, etc. It also contains a 16-bit CRC for validation. One of the following occurs:
 - a. If either the secure hash or TOC2 validation fails, CM0+ moves to DEAD protection state and remains in a continuous loop until the device is reset. This guarantees that only verified code will be executed, and no user code will be executed if there is a possibility that the device has been compromised.
 - b. If the secure hash and TOC2 are validated, the debug access ports are configured to the values stored in the SARs bytes in the eFuse. The GPIO pins used for the debug port on the device are left in their default tristate mode and will not communicate to the debugger or programmer in this state, even if the SAR defines all ports to be open. The 1 M/512 K/256 K flash parts provide an option in the TOC2 to automatically configure the debug GPIO for debug operation. This allows you the flexibility to configure the secure hardware before the debug ports allow a connection. Your application can control the access dynamically if need be. Example code to enable the debug port is presented later in this document.

Note: The SAR bytes, public key, and TOC2 are written into the device prior to moving from NORMAL to SECURE lifecycle stage. Once in the SECURE lifecycle stage, these values cannot be modified.

5. The system has now been fully validated and the debug modes have been configured to the designer's requirements. A header prefix (see [section 4.2](#)) was added to the user code that contains information such as number of CPUs (2 for PSoC™ 62/63) and the starting location for each CPU's application code. Flash boot checks this header to determine the location where the user CM0+ code starts.
6. CM0+ jumps to the CM0+ user project.

The CM0+ user code that is first executed after Flash boot does not have to be the main application. It could in fact be a bootloader that manages updates for either CM4 or both CM0+ and CM4. Also, this may be a good place to implement your application-level security by programming protection units and the protection context. See [Appendix E - Protection unit configuration](#) for more details.

5 PSoC™ 6 application notes

5.15.3.4.3 SECURE_WITH_DEBUG lifecycle stage

This lifecycle stage operates just as the SECURE lifecycle stage, except that it uses NARs so that you can debug your device before moving to SECURE mode. Once you have moved to SECURE_WITH_DEBUG mode, you CANNOT move back to NORMAL or to SECURE lifecycle stage. Devices in this stage should never be shipped to the end customer. NORMAL lifecycle stage with validate is a much better step to validate your key generation and code signing. In SECURE_WITH_DEBUG, you can change your application, but not SFlash which includes the private key and TOC2.

5.15.3.4.4 NORMAL lifecycle with validate

This is the NORMAL lifecycle stage with the option to validate the first code. It operates just as the SECURE lifecycle stage, but skips the secure hash validation, uses NAR instead of SAR, and configures the debug GPIOs for communication with the debug hardware before executing the user code.

This lifecycle stage is useful to verify that your key generation and code signing process is working properly. If you find a problem, it is easy to debug, and you can erase the entire device and start over if a problem is found. Make sure that you have not set the NAR to be too restrictive or enabled it at all so you can debug the problem. Once your key generation and code signing are validated, you are much less likely to have problems switching to the SECURE lifecycle stage. To enable code validation, you must create a valid TOC2 and set the appropriate bits in the Flash boot parameters. (See [section 4.1](#) for details of TOC2.)

5.15.3.4.5 Debug boot errors

If there is an error during the boot sequence in NORMAL or SECURE lifecycle stage, the device will enter the DEAD state in which CM0+ stays in an endless loop. If the device was in the SECURE lifecycle stage, the device will change the protection context to PC=2. If in the NORMAL lifecycle stage, the PC value remains at PC=0.

Whether you can debug the device or not depends on your settings to access in the DEAD state. During debugging, you should leave full access to the debug ports. To determine what caused the boot sequence to fail and enter the DEAD state, read the value of IPC (#2) data register. An error code with the failure ID will be written into that register. See [Appendix E - Protection unit configuration](#) for boot sequence error table.

5.15.3.5 Chain of Trust (CoT)

At the beginning of the Chain of Trust, must be immutable code that cannot be altered. This first immutable code is referred to as the “Root of Trust” (RoT). The initial ROM code validates the Flash Boot code in SFlash to verify that it hasn’t been modified, prior to any code execution in SFlash.

Flash boot, trim constants, and the Table of Contents1 (TOC1) are located in SFlash (Supervisory Flash) and are restricted from being reprogrammed in either NORMAL or SECURE lifecycle stages.

The SFlash area is validated with a Factory_HASH value stored in the eFuse. The Factory_HASH code is not used during the boot sequence in NORMAL or SECURE lifecycle stages, but is used to validate the part before moving to the SECURE lifecycle stage. This ensures that the Flash boot code, trim values, and the TOC1 has not been tampered with after the MCU has left Infineon. If the Factory_HASH has been corrupted, Infineon should be contacted immediately. See [Appendix F - Debug codes for failed boot sequences](#).

After the transition from NORMAL to SECURE lifecycle stage, all blocks in the SFlash, including the public key area and the TOC2, are validated with the Secure_HASH each time the device boots. This secure hash is stored in the eFuse and cannot be changed without detection. If an error is found while validating the SFlash, the device will abort the boot sequence and enter a DEAD state. [Figure 545](#) shows the CoT from the perspective of data and code validation.

5 PSoC™ 6 application notes

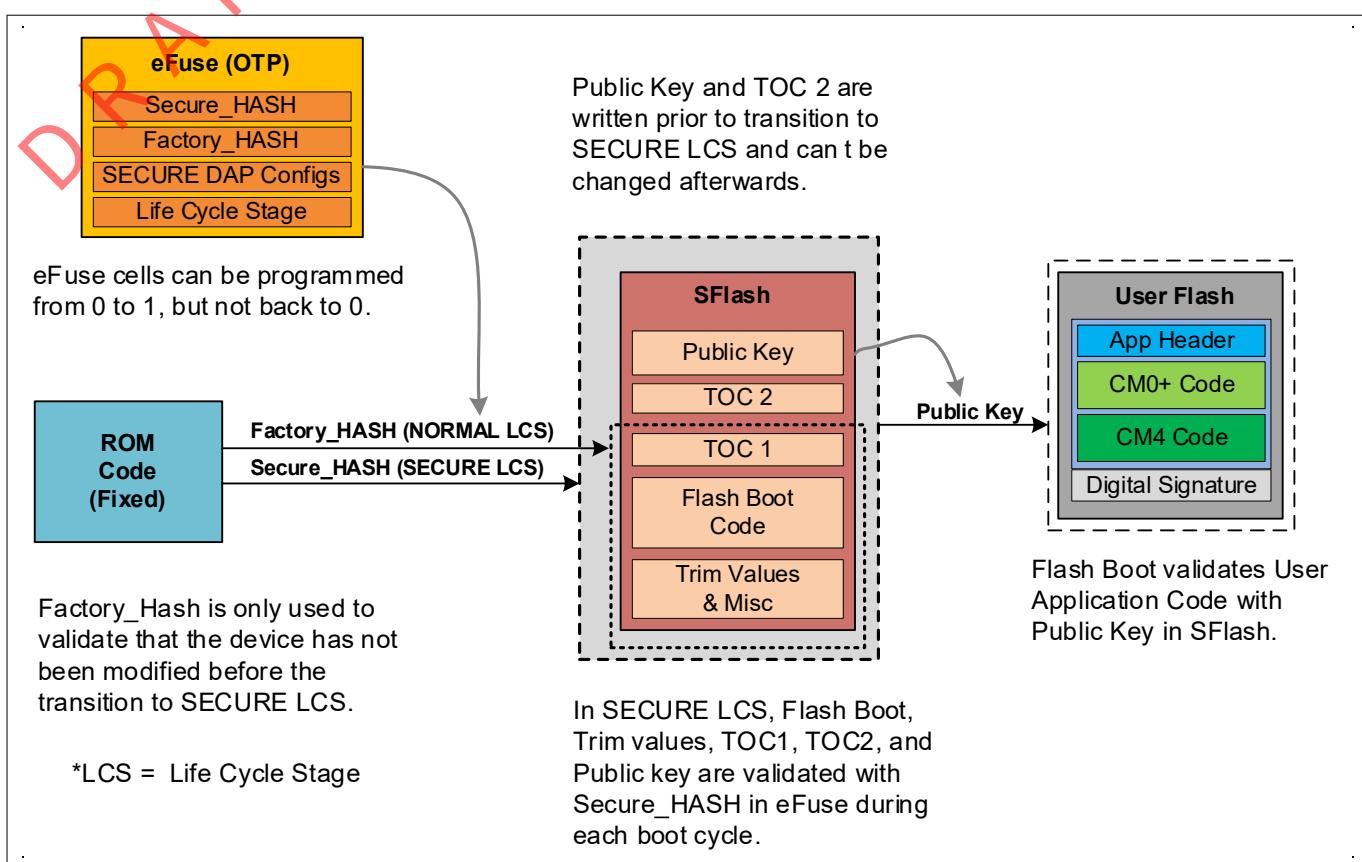


Figure 545 Basic chain of trust

At this point, the entire SFlash is now trusted because its validation is based on the memory (eFuse) that cannot be modified without detection during SFlash validation in the ROM.

The (OEM) public key, which is locked into the SFlash, is secure and cannot be changed without being detected as well. It is used by Flash boot to validate the next step in the boot process. Flash boot validates the code in the user application block, which includes a digital signature at the end of the code block. Flash boot uses the SHA-256 hash function to calculate the digest of the user application. The digital signature attached to the user application is encrypted using a private key that is associated with the public key stored in the SFlash. The encrypted hash uses RSA 2048-bit encryption.

The calculated and the stored digest (decrypted digital signature) are then checked to see whether they match. If they match, the user application has been verified.

5 PSoC™ 6 application notes

5.15.3.6 ~~Code signing and verification~~ Code signing and verification

In the [Chain of Trust \(CoT\)](#) section, code verification was mentioned but with not much detail. In this section, we will dive into the process of signing a block of code so that it can be verified during runtime. Infineon supports the signing of application code using the CyMCUElfTool which is downloaded with ModusToolbox™ software.

The encryption method used is PKC (public key cryptography) that has a private and public key pair. You must ensure that the private key is kept in a secure location, so that it never gets into the public domain. If the private key is exposed, it will endanger your system's security. Companies must create a method in which very limited access to the private key is allowed.

The public key, on the other hand, can be viewed by anyone. The only requirement is that the public key must be validated or locked in such a way that it cannot be changed, or so that any modification to the public key can be detected. In this example, the public key is stored in the SFlash and verified with the Secure_HASH as defined in the [Chain of Trust \(CoT\)](#) section. These private and public keys can be generated with common encryption libraries such as OpenSSL.

Do the following to transition a PSoC™ 62/63 device to the SECURE lifecycle stage and use code signing:

1. Fill in TOC2 and add the 16-bit CRC at the end of TOC2 (SFlash).
2. Fill in the RSA-2048 public key (SFlash).
3. Use standard CYPRESS™ application format (user flash) at the beginning of the application (See [section 4.2](#) for definition).
4. Sign your application bundle and place the 256-byte (2048 bits) digital signature at the end of the code (user flash).
5. Enable NORMAL lifecycle code verification or move to SECURE lifecycle stage.

It is HIGHLY recommended to test the code verification in the NORMAL lifecycle stage before switching to the SECURE lifecycle stage. In the NORMAL lifecycle stage, if something is incorrect with the code signing process, you can reprogram the TOC2, public key, and update your application. Once you switch to the SECURE lifecycle stage, you cannot alter the TOC2 or update the public key. You may not be able to change your application with the programmer if you closed your debug ports and your bootloader is not yet implemented.

For more information on generating and using the private and public keys, see [Appendix A - Code example of a security application template](#).

5.15.3.6.1 ~~Code signing~~ Code signing

To verify the user application, a digital signature is created and appended to the end of the code during build time. The code itself is not encrypted but the digital signature is the encrypted digest. The digital signature is generated by encrypting the digest with the RSA-2048-bit algorithm. The digest is generated by running the user application binary through a SHA-256 hash function. This type of code signing is used for both RSA and ECDSA algorithms, but PSoC™ 62/63 Flash boot code only supports RSA-2048. This method guarantees that a third-party without access to the OEM's private key cannot properly sign the application code (see [Figure 546](#)).

5 PSoC™ 6 application notes

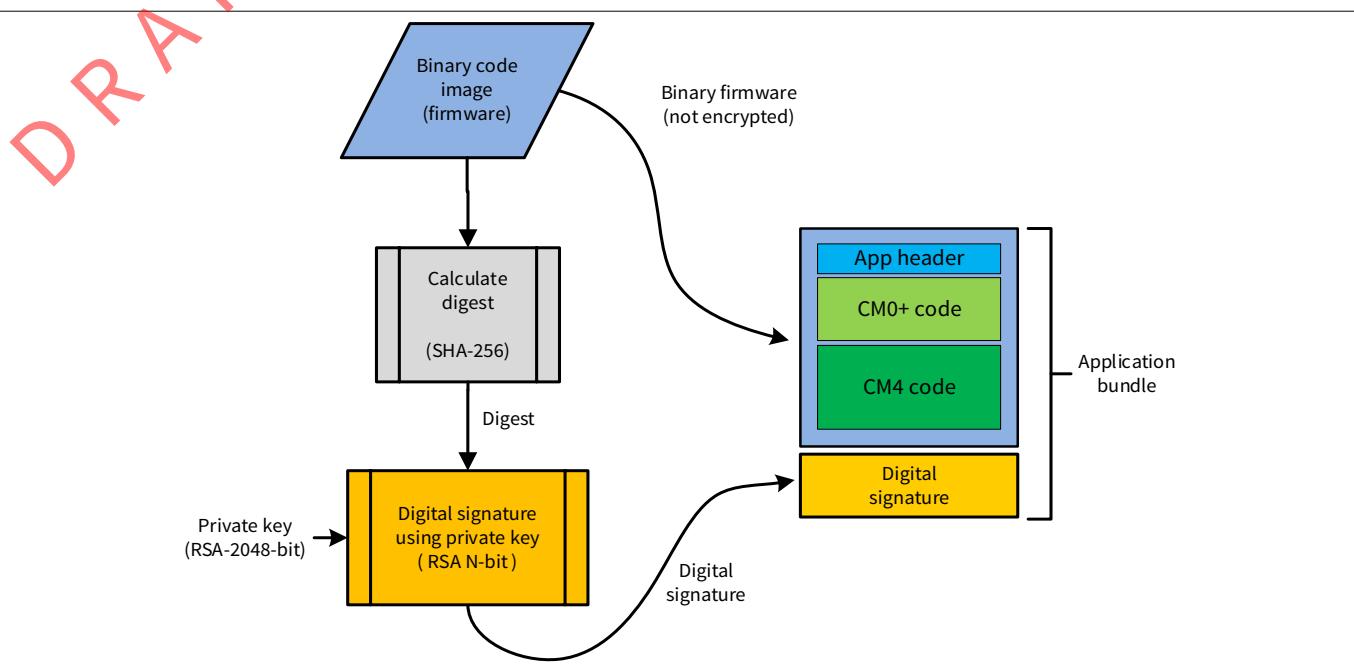


Figure 546 Generation of digital signature

5.15.3.6.2 Code verification

A secured system must be able to verify that its application code was indeed generated from a known source and detect if it has been modified by a third-party or corrupted. If the verification fails, the execution must take a known path to a safe state.

Parts of code validation

Verification requires three parts: application code, digital signature, and key pair (public and private). The application code and digital signature come as a pair; the public key is stored in the device such that it cannot be changed, without detection.

- **Application code:** This includes both executable code and constants that make up the firmware in an embedded system. This code usually resides in the flash memory that can be modified at one time or another. Therefore, you must be able to determine whether this code is from a known source (OEM) and has not been corrupted either by accident or by a malicious event.
- **Digital signature:** A digital signature is the encrypted digest (hash) of the application code generated at the OEM. The digest of the application code is encrypted by the private key to generate the digital signature. The hash algorithm used in this case is SHA-256. The digital signature is then used to verify that the application before being executed.
- **Key pair:** This asymmetric key pair contains both the private and public keys. In the system described in this application note, the public key is stored on the device and the private key is secured by the OEM. The public key must be secured in one of two ways (the second option is the most likely one for an embedded system):
 - A method to verify the source of the key. This can be accomplished with some type of communication with a known source or server. This is not practical for devices that cannot easily communicate with a known server when required.
 - Have the key protected such that it cannot be changed, or that you can determine if it has been modified. In the PSoC™ 6 devices, a hash is calculated from the areas containing the public key, Flash boot code, and trim values. This hash is then stored in one-time programmable eFuse and referred to as Secure_HASH. The hash is verified before the public keys are used.

~~5 PSoC™ 6 application notes~~

Code verification used by PSoC™ 62/63 MCUs

The device bootup code “Flash boot” uses RSA to verify the first user code. The application binary code block (application bundle) includes a digital signature that is created during the build time. To verify the application code, a hash function (SHA-256) is calculated across the binary code block. Next, the digital signature is decrypted using the stored public key to reveal the original digest generated when the application was generated at the OEM location. The calculated digest and the decrypted digest (from digital signature) are compared to verify they are equal. If they are an exact match, the code is verified (see [Figure 547](#)).

ECDSA is another common algorithm used to verify application code. It is similar to RSA, but instead of decrypting the digital signature and comparing it to the calculated hash, the calculated digest, digital signature, and ECDSA public key are used to generate a pass or fail. This is a common method used by MCUboot to verify the application code. [Figure 547](#) shows the comparison between RSA and ECDSA.

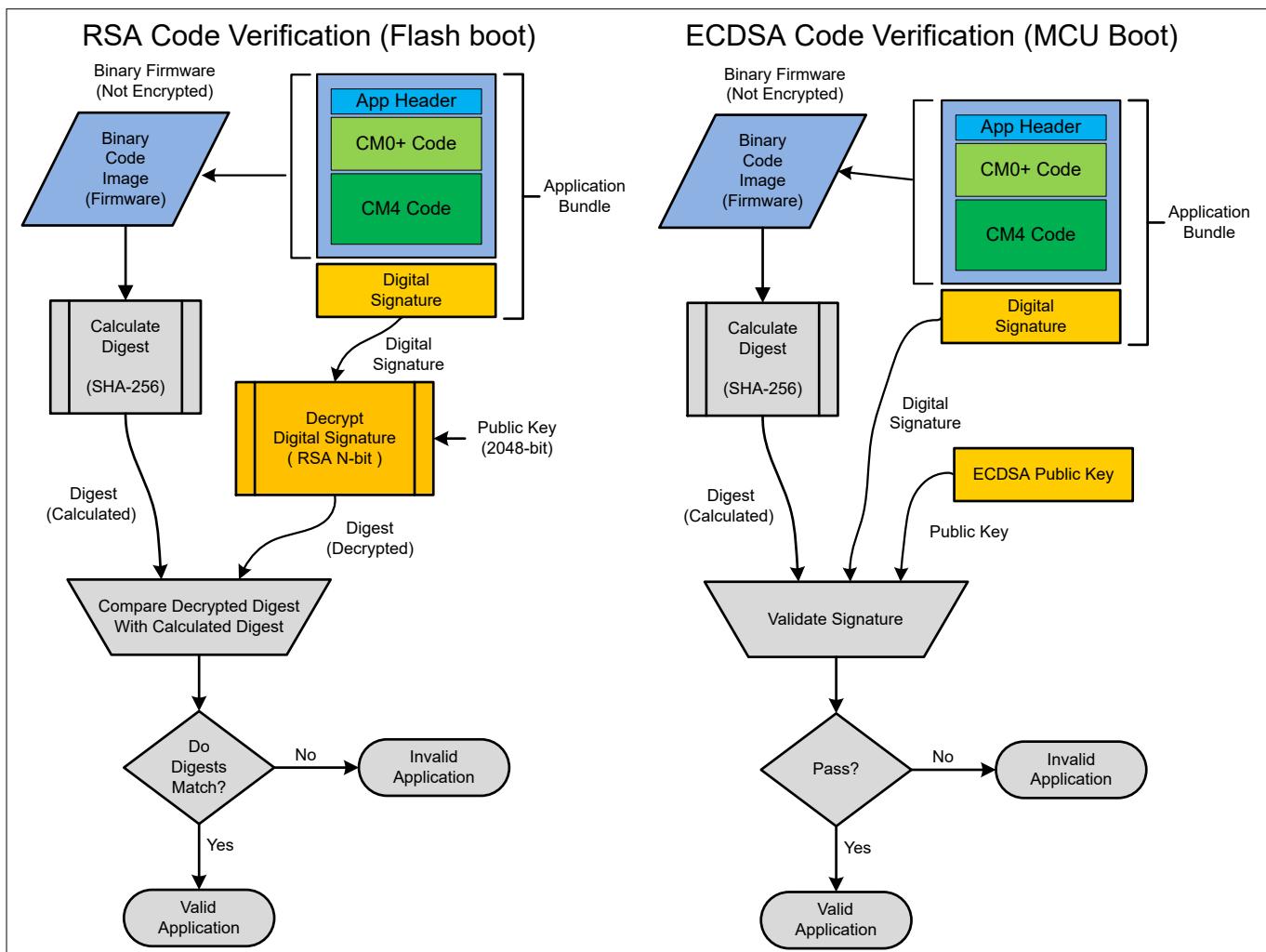


Figure 547 Application verification

5.15.3.7 Protection units

This section provides a brief description of how the protection units work and how to use them. For a more detailed description of protection units, read the “Protection Units” chapter in the [PSoC™ 6 MCU Architecture TRM \(Technical Reference Manual\)](#) for your device.

PSoC™ 6 has several bus masters that all share a single bus that interconnects the flash, SRAM, ROM, GPIOs, and peripheral control registers. The bus masters include the two CPUs (CM0+ and CM4), DMA controllers (Data

~~5 PSoC™ 6 application notes~~

Wire), crypto unit, and the test controller. Bus arbitration hardware keeps the bus masters from bumping into each other, but the protection units keep them from reading and writing into each other's memory space.

Protection units can be programmed to allow only specific CPUs and other bus masters to access only predefined memory segments and peripherals. If a bus master attempts to use a section of memory not allowed by the protection unit, a bus fault will occur. Because the two CPUs (CM4 and CM0+) share the same memory space in the PSoC™ 6 MCUs, the protection units guarantee that one CPU cannot affect the memory or peripherals allocated to the other if needed. Not all memory or peripherals must be allocated to just one CPU; by default, the entire memory space is shared between the two CPUs and sections can remain that way if desired.

Protection units can also be used to isolate memory for different tasks that run concurrently in an RTOS. Different sections of a boot process may also be another example of securing sections of memory. For example, you may want your bootloader to have access to write to the flash in the user application area, but deny the user application from overwriting writing code that is executing. Protection units can be configured to do just that. [Figure 548](#) shows how the different protection units (SMPU, MPU, PPU) are connected relative to the bus masters, peripherals and system bus (AHB).

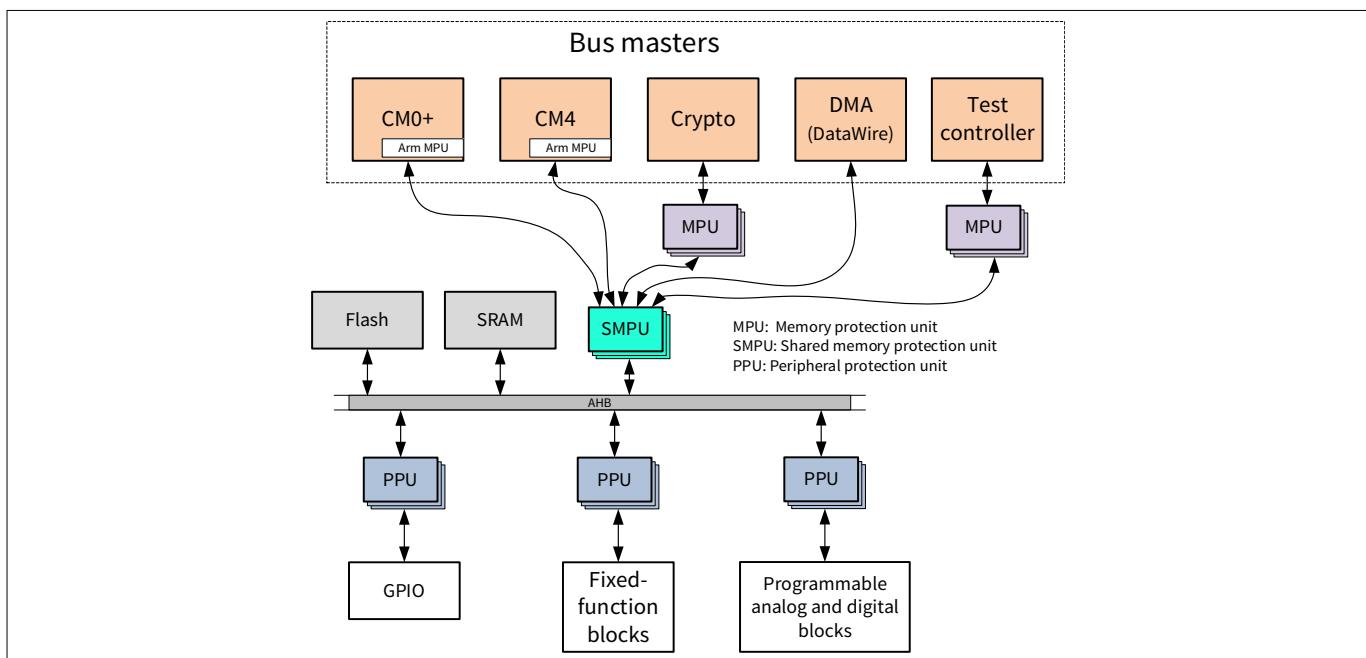


Figure 548 Protection units with respect to the overall system

PSoC™ 6 has four types protections units:

- SMPU – Shared memory protection units
- MPU (Arm®) – Memory protection unit that are part of the CPU IP
- MPU (Infineon) – Memory protection unit for test controller and crypto block
- PPU – Peripheral protection unit (fixed and programmable)

5.15.3.7.1 SMPUs

The SMPUs are capable of protecting any section of the bus memory space, but they are meant to be used for segments of the ROM, SFlash, user flash, and SRAM. The CPUs (CM0+ and CM4) each contain an Arm® MPU that is part of the Arm® IP. The Arm® MPUs do not support protection context (**will discuss this later**), secure, or privileged modes. but they are primarily meant to be used in conjunction with an RTOS or other operating systems.

Two Infineon MPUs are used in the crypto block and the test controller (debug interface). For more information on these MPUs, see the Technical Reference Manual (TRM) for the device you are using.

~~5 PSoC™ 6 application notes~~

~~5.15.3.7.2 PPUs~~

~~DRAFT~~ There are two different types of PPUs: fixed and programmable. Fixed PPUs are assigned a specific block of registers, usually for a single IP block such as a TCPWM, SCB, or SAR ADC. The address and region size are fixed and cannot be changed.

Programmable PPUs are similar to the SMPUs in that they can protect any region of peripheral device address space. They are meant to protect registers in a specific block that are not covered by the resolution of the fixed PPUs.

5.15.3.7.3 Protection unit configuration

Protection units have several parameters, besides just address and range. The following is a list of attributes that must be met when evaluating if the memory access is permitted by a bus master.

Table 122 Protection unit parameters

Parameter	Description
Address	Where the region starts. It must be aligned to the region size.
Region size	Size of the region which is a power of 2 from 256 bytes to 4 GB.
Sub regions	Disables any of 8 equal spaced regions. If Region size is 256, each sub region would be 32 bytes, each which can be disabled.
User permission	Read/Write/execute for user level access.
Privilege permission	Read/Write/Execute permission for CPU privilege mode.
Secure	This bit is not directly associated with the NORMAL/SECURE lifecycle stage; it can be set for a bus master. You can use this bit to designate one of the bus masters as the “secure” processor, most likely the CM0+. This bit adds one more level of designation. For example, you could have two bus masters (CM0+ and CM4) both set to protection context equal 1, but only the designated secure bus master may have access to a region designated by a protection unit.
PcMatch	This bit is valid only if two or more of the protection structures match the same address range that the bus master is attempting to address. If two or more protection structures match the bus master’s address request, the protection system will evaluate these structures from highest index (15) to lowest index (0). It will continue to evaluate these structures until it finds PC_MATCH = 0, at this point it will complete the evaluation of the present structure and not evaluate any further structures, even if the address or PC values match the request. If the present protection structures PC_MATCH = 1, it will continue to the next structure that has a valid address range.
PcMask	Bitmap of what PC value may access memory region defined by this protection unit.

All the protection units, except for the MPUs that are part of the Arm® CPUs, have a master protection structure associated with it. The master protects the PPU (slave) from being altered by any bus master that should not have access. In a sense, the master assigns an owner which is the protection context. See the next section for more information.

Although the two CPUs share the same bus, they run totally independent code. The code running in the two CPUs can even be written by two different companies, depending on how you structure your device. For example, the system calls in the PSoC™ 6 are written by Infineon, but the rest of the code will be written by the customer.

5 PSoC™ 6 application notes

The bootup ROM enables protection units to protect the stack area and the registers that control writes to the flash and eFuse. This is important for the following reasons:

1. Prevent unintended modifications to the flash by the code in SECURE mode.
2. Protecting the flash control hardware forces the user to use system calls to write to the flash which is guaranteed to optimize lifecycles with maximum retention.
3. Protected system calls adhere to the SMPU settings to determine who has permissions to change the flash in SECURE and NORMAL modes.

A common configuration for a secure system is to have CM0+ be the secure processor and CM4 be the main (non-secure) application processor. The two CPUs should operate independently of each other with only a controlled interface between them.

Using protection units, the memory spaces can be totally isolated from each other although they share the same memory bus. Peripherals and GPIOs, such as flash control, can be reserved by one CPU so it cannot be accessed by the other CPU. The controlled interface between the two CPUs as mentioned before can be the shared SRAM or in the case of system calls, a combination of shared SRAM and the IPC interface.

5.15.3.7.4 Bus masters

In PSoC™ 6 MCUs, a bus master is any block that can directly access the SRAM or flash without the aid of another bus master. There are at least six bus masters in the PSoC™ 6 MCUs, and maybe more in future devices. In this document, a mention of bus master includes the CM0+ and CM4 processors. [Table 123](#) shows the bus masters and important configuration registers.

Table 123 Bus masters in PSoC™ 6 MCUs

Master #	Bus master	Master Protection Context Control register	Master Control register
0	CM0+ processor	PROT_SMPU_MS0_CTL	PROT_MPU0_MS_CTL
1	Crypto block	PROT_SMPU_MS1_CTL	PROT_MPU1_MS_CTL
2	DataWire 0 (DMA)	PROT_SMPU_MS2_CTL	Inherits settings from the CPU
3	DataWire 1 (DMA)	PROT_SMPU_MS3_CTL	Inherits settings from the CPU
14	CM4 processor	PROT_SMPU_MS14_CTL	PROT_MPU14_MS_CTL
15	Test controller	PROT_SMPU_MS15_CTL	PROT_MPU15_MS_CTL

The Master Protection Context Control registers (PROT_SMPU_MSx_CTL) are key to making protection units function. They are used to configure each bus master with the appropriate attributes that the protection units use to determine access. The protection unit library contains the `cy_Port_ConfigBusMaster()` function that can be used to configure these registers.

The Master Control register for each bus master controls the active protection context. The DataWire blocks do not have an associated register because they inherit their attributes from the CPU. The protection unit library (PROT) includes the `cy_Prot_SetActivePC()` function to set the active protection context for the bus master. This function will only allow you to set a legal protection context for the device. Each device has a mask register that determines which PC (protection context) value can be assigned for that bus master. Documentation for all ModusToolbox™ (PDL) libraries may be found at [PSoC™ 6 Peripheral Driver Library](#).

5.15.3.7.5 Protection contexts (PC)

Understanding how a protection context (PC) works is mandatory to configure the protection units. A protection context is similar to groups in a computer system while the bus masters are users in those groups. Each bus master (user) can belong to a subset of all possible PC values (groups), but can only be assigned one PC value at a time. PSoC™ 6 family devices support PC values of 0 through 7.

~~5 PSoC™ 6 application notes~~

A bus master has a mask of what PC values it can be assigned, and the current PC value. At any time, the bus master can change its current PC value to one enabled in the mask, not to a PC value not in the mask. If a bus master's PC mask enables PC values of 1, 2, and 3, and an attempt is made to change the PC value to PC=4, an error will occur and the PC value will not be changed. Without this feature, any bus master could just change its PC to any value.

A PC value of 0 is a special case. If a bus master has a PC=0, it may access any memory location without causing a page fault no matter how the protection units are configured. Think of it as super user mode. By default, all bus masters are set to PC=0 after reset. After all protection units are configured, all bus masters should be changed to a PC value other than 0 to make sure that the protection/security configuration cannot be altered.

Protection units are not assigned to bus masters, but instead you select which protection contexts are valid to access the region specified by the protection unit. Each protection unit has a PC mask that determines which PC values are accepted. During configuration, determine which PC values should have access to the region specified by the protection unit.

For example, if an SMPU was set up with a PC mask that allowed PC values of 4 and 5, bus masters with a PC value 4 or 5 that met all other criteria could access the region specified by the SMPU. A bus master with a PC=1 cannot access the region specified by this SMPU no matter it met all other criteria.

One or all bus masters can share the same protection context, or they may all be different. For example, sharing the SRAM with everyone but blocking any CPU from executing the code in the SRAM. A bus master with a PC=0, always has full read/write/execute to the entire memory space no matter the configuration of any protection units.

Although a bus master PC mask may allow several PC values, the currently selected PC value is the only one that matters when accessing a protected memory area. For example, CM4 (bus master) had a PC mask that allowed PC values 1, 2, and 3 and had a current PC value of 3. If it attempted to access a region protected by an SMPU that only allowed 1 or 2, the access would fail and cause a hard fault because the bus master's current value was not 1 or 2. If CM4 had changed its protection context to 1 or 2 before accessing the region, the operation would work without error.

CM0+ is the only bus master that can switch back to PC=0. The system calls revert CM0+ back to PC=0 each time the system call ISR is executed. It does this by setting up CPUSS_CM0_PC0_HANDLER with the address of the system call ISR. When one of the three IPC channels (0, 1, or 2) causes an NMI (non-maskable interrupt) interrupt, CM0+ is automatically switched to PC=0, where it will execute the system call and then revert to the original PC value upon return of the interrupt. 1st generation parts can do this just for PC=0, but 2nd generation parts can be configured to automatically switch to PC values 1, 2, and 3 with an ISR.

See register description in the respective TRMs for CPUSS_CM0_PC1_HANDLER, CPUSS_CM0_PC2_HANDLER, and CPUSS_CM0_PC3_HANDLER.

5.15.3.7.6 SMPU/PPU master

Protection units are divided up into a slave and a master. The slave defines the memory or register space, protection parameters, and what PC values have access. The master protects the slave structure from accidental or intentional modification of the slave structure. There is always one master for each slave protection unit structure and therefore has a fixed address and region area that is only readable. After all slave structures are configured, it is best to own all masters by a single PC value, such as PC=0. This way after the bus masters have all been changed to PC ≠ 0, the system protection unit configuration is fixed.

5 PSoC™ 6 application notes

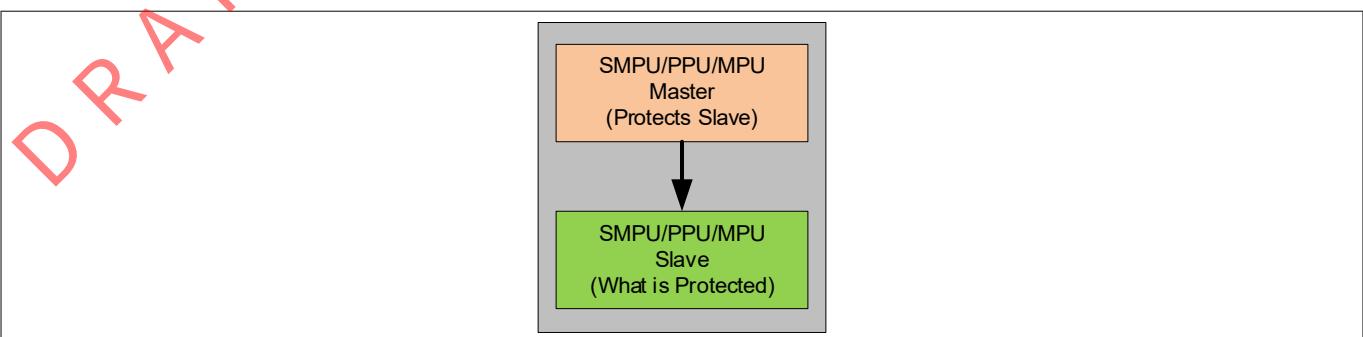


Figure 549 Protection unit slave/master

5.15.3.8 Debug port configuration

See [Appendix C - Debug port access settings](#) for register and bit level definitions.

Attention: Configuration of the debug ports and the lifecycle stage consists of programming the system eFuse bits. The eFuse bits are one-time programmable and once set cannot be erased. This means that if you disable the debug ports by changing the lifecycle stage, there is no way to change it and you cannot re-enable the debug ports. If a bootloader has not been installed, you will not be able to program the device or update the code.

Attention: ALWAYS program your application into the device before changing the debug ports or changing the lifecycle eFuse bits. If you are using a bootloader, verify its operation thoroughly before changing these eFuse bits. Once you advance to SECURE mode, there is no going back, and you cannot change your eFuse settings afterwards.

5.15.3.8.1 Debug port architecture

The physical interface to the debug port is the same as many other Arm®-based devices. It consists of either 3-pins for SWD, serial wire debugger (SWDCLK, SWDIO, nTRST) or 5-pins for JTAG (TMS, TCLK, TDI, TDO, nTRST). When the debugger is connected, it negotiates whether it will communicate via SWD or JTAG. Most of the development tools use the SWD interface.

There are three debug ports on the PSoC™ 62/63, CM4 CPU access port (CM4-AP), CM0+ CPU access port (CM0+_AP), and the system access port (SYS_AP). The CM4-AP and CM0+_AP are primarily used for debugging the individual CPU. When connecting to either of the CPU access ports, the debugger has access to the CPU's debug registers such as the breakpoint registers and has full access to anything that the CPU has access to through the CPU. The SYS-AP does not have access to the CPU debug registers, but may access any memory or registers in the memory map. The debugger/programmer may connect to any of the three debug ports.

5 PSoC™ 6 application notes

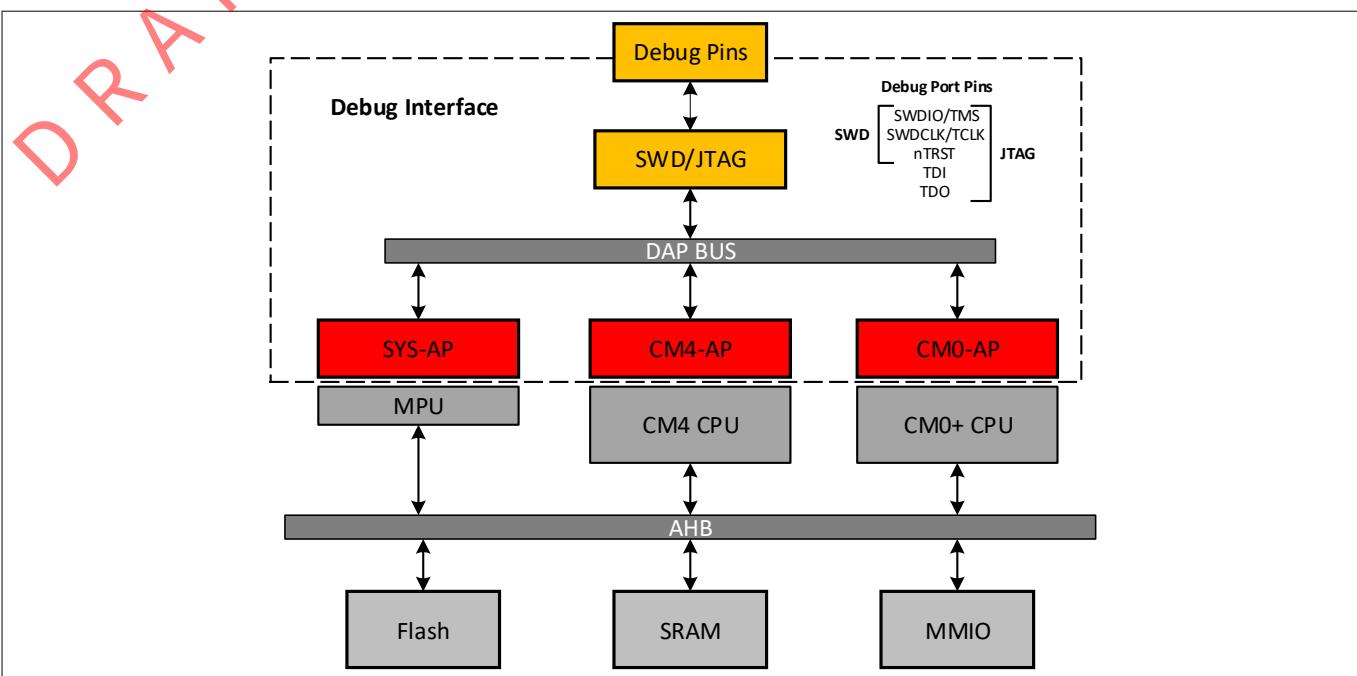


Figure 550 PSoC™ 6 debug port hardware

Any combination of these debug ports may be enabled or disabled. When talking about the debug ports, disabling the port is not the exact opposite of enabling the port. There are individual bits that control “Disabling” and “Enabling” the port. When a port is disabled, it is done during bootup in the ROM and the bit cannot be changed via the debug interface or the internal CPUs. This ensures that if a debug port is disabled, it cannot be re-enabled accidentally or maliciously.

In the NORMAL lifecycle stage, if the ports are not disabled, they will be automatically enabled, and the GPIO pins will be configured to interface with a debugger. In the SECURE lifecycle stage, if you do not disable the debug ports, the debug ports will be enabled, but the GPIO pins used for debugging will not be configured to interface with a debugger, you must do that in the application code. The reason for this is so that the user application can determine during runtime, when or if the access ports will be available. This adds another level of flexibility to the SECURE lifecycle stage. There is an option for the 1st generation parts to automatically configure the GPIO pin, see [section 4.1 Table of Contents2](#).

There are three different access port restriction settings:

- Secure access restrictions (SAR)
- Normal access restrictions (NAR)
- Dead access restrictions (DAR)

The NAR and SAR are used to set up the debug port restrictions for SECURE and NORMAL lifecycle stages respectively. The DAR are used in the SECURE lifecycle stage if there is an error during the secure boot process. This could occur if the memory has been corrupted, an incorrect public key is used, the code is signed incorrectly, or the TOC2 is corrupted.

The SAR and DAR are stored in the eFuse which cannot be erased once set. These restrictions must be set before entering the SECURE lifecycle stage. Once in SECURE lifecycle stage, the SAR and DAR cannot be altered.

The NARs are stored in the SFlash and are protected only by the system call firmware which ALWAYS runs with a protection context equal 0. The system call functions allow you to increase the NAR security, but not to reduce it. It is possible to write code to bypass the system call functions to reprogram the SFlash where the NAR are stored. To prevent this, make sure that all bus masters, including CM0+ and CM4, have their protection context set to other than the default of 0. Protection units have already been configured to protect the Flash programming registers, so by changing the protection context to other than 0, only the system calls can modify the SFlash or flash, and these functions check protection unit settings.

~~5 PSoC™ 6 application notes~~

NAR are not recommended for a truly secure system, but can be sufficient for many use cases if used correctly. One advantage to using the NAR is that it is stored in the SFlash, you do not need a 2.5 V supply on VDDIO0 to program them.

~~DRAFT~~ **Table 124 Access restrictions**

Access restrictions	Where stored	Immutable?	Active lifecycle stage	Notes
Normal (NAR)	Protected SFlash	No	NORMAL	May be altered with internal firmware but can be protected using protection context.
Secure (SAR)	eFuse	Yes	SECURE	Cannot be altered.
Dead (DAR)	eFuse	Yes	SECURE	Cannot be altered; not valid in NORMAL lifecycle stage These restrictions are implemented when the device does not boot properly, such as corrupt TOC2 CRC, or invalid application signature.

Note: *The debug ports should be totally disabled when in the SECURE lifecycle stage for the best security.*

5.15.3.8.2 System access port (SYS-AP)

The SYS-AP is notably different than the CM0-AP and CM4-AP. It provides direct access to all system memory and memory-mapped I/O (MMIO) by default. A programmable memory protection unit (MPU) is attached between the SYS-AP and the AHB. It can be configured to limit access to sections of the flash, SRAM, and MMIO registers.

By default, this MPU is disabled, but you can enable it and provide limited access to the memory instead of all or nothing. Both the flash and SRAM are configured the same way as a portion of the available memory range, starting at the lowest address. See [Appendix C - Debug port access settings](#) for register and bit level definitions.

~~5 PSoC™ 6 application notes~~

~~5.15.4~~ Project configuration

As you see, building a fully secure system with a CoT is more complicated than generating a simple application. Instead of just the user code, the hex file must contain several other pieces of data/code normally not required in a simple (non-secure) system. The following are the memory sections that will need to be programmed when creating a secure system with application authentication.

- Lifecycle stage and DAP configurations (eFuse)
- Public key (SFlash)
- TOC2 (SFlash)
- CYPRESS™ application header (user flash)
- User application block (user flash)
- Digital signature (user flash)

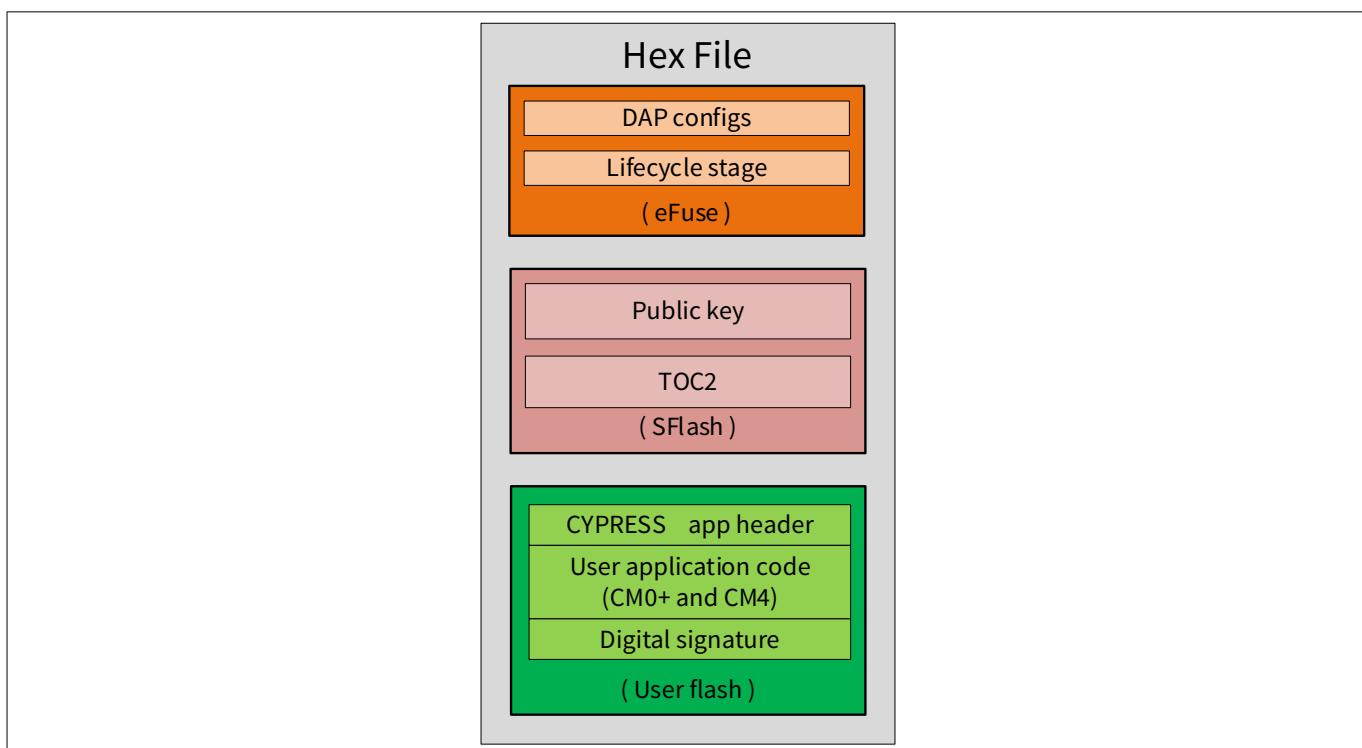


Figure 551 Secure system hex file configuration

The linker files in ModusToolbox™ define the memory locations specifically for the public key and TOC2 as well as the NAR. This makes it easy to place the data where it is expected in memory. The code snippet below is from a GNUGCC linker file.

```
sflash_nar      (rx) : ORIGIN = 0x16001A00, LENGTH = 0x200 /* SFlash: Normal Access
Restrictions (NAR) */
sflash_public_key (rx) : ORIGIN = 0x16005A00, LENGTH = 0xC00 /* SFlash: Public Key */
sflash_toc_2      (rx) : ORIGIN = 0x16007C00, LENGTH = 0x200 /* SFlash: Table of Content # 2 */
sflash_rtoc_2     (rx) : ORIGIN = 0x16007E00, LENGTH = 0x200 /* SFlash: Table of Content # 2
Copy */
```

~~5 PSoC™ 6 application notes~~

~~5.15.4.1~~ Table of Contents2 (TOC2)

~~TOC2~~ is used to point to the location of the first and second executable applications. It has a redundant copy (RTOC2), each with its own CRC for validation. Each copy of TOC2 is on a separate flash page so that if one is corrupted during writing, it is unlikely that the other is corrupted. Flash boot uses the first one with a valid CRC. If both fail CRC validation, Flash boot will remain in a loop, ready to be reprogrammed if in the NORMAL lifecycle stage, or in a DEAD state if in the SECURE lifecycle stage. See the code snippet below for the definition of the TOC2 structure.

```

/* Table of Content structure */
typedef struct{
    volatile uint32_t objSize;      /* Object size (Bytes) */
    volatile uint32_t magicNum;     /* TOC ID (magic number = 0x01211220 ) */
    volatile uint32_t userKeyAddr;  /* Secure key address in user Flash (optional) */
    volatile uint32_t smifCfgAddr;  /* SMIF configuration structure (optional) */
    volatile uint32_t appAddr1;     /* First user application object address */
    volatile uint32_t appFormat1;   /* First user application format */
    volatile uint32_t appAddr2;     /* Second user application object address (optional) */
    volatile uint32_t appFormat2;   /* Second user application format (optional) */
    volatile uint32_t shashObj;     /* Number of additional objects to be verified(Secure-HASH)
*/
    volatile uint32_t sigKeyAddr;   /* Signature verification key address */
    volatile uint32_t addObj[116];  /* Additional objects to include in Secure-HASH */
    volatile uint32_t tocFlags;    /* Flags in TOC to control Flash boot options */
    volatile uint32_t crc;        /* CRC16-CCITT */
}cy_stc_ps_toc_t;

```

Table 125 TOC2 parameter definitions

Parameter	Size	Description
objSize	32-bit number	This is the flash row size (512 bytes) minus the size of the crc (4 bytes), which is 508.
magicNum	32-bit value	This is a magic number to help to verify the structure quickly. A valid TOC2 will always have the value <u>0x0121_1220</u> .
userKeyAddr (optional)	32-bit value	This is a pointer to an optional area of additional key storage. This is optional and may be zero. If used to store keys that should not be changed, it should be added to the blocks that are hashed.
smifCfgAddr	32-bit Address	Null terminated table of pointers representing the SMIF configuration structure.
appAddr1	32-bit Address	This is a pointer to the first user application. In the code example application associated with this application note, it will be the pointer to the bootloader project. If TOC2 is invalid, Flash boot assumes the starting code is at <u>0x1000_0000</u> .
appFormat1	32-bit value	This is the format of the header for the project pointed to with “appAddr1”. Use only CYPRESS™ application format, all other formats have been deprecated.

(table continues...)

5 PSoC™ 6 application notes

Table 125 (continued) TOC2 parameter definitions

Parameter	Size	Description
appAddr2 (optional)	32-bit Address	This points to an optional application address that Flash boot does not use. The user could have the bootloader refer to this address as where the actual application starts. In this example, this value is null.
appFormat2 (optional)	32-bit value	This defines the appFormat for the application pointed to by appAddr2. In this example, this value is null.
shashObj	32-bit number	The number of objects in addition to the objects included in the FACTORY_HASH, starting with “sigKeyAddr” that are listed in “addObj” that will be included in the SECURE_HASH. In the SECURE LCS all these items will be validated at boot time by default. The maximum number of items is 15. If no additional objects are hashed, this value should be “1” for the RSA public Key.
sigKeyAddr	32-bit Address	This is a pointer to the RSA public key that is used to validate the first project pointed to by “appAddr1”.
addObj	[116] 32-bit Addresses	This is an array of pointers to objects, terminated by a null value, that will be added to the SECURE_HASH. This array may be up to 15 words long and the remaining values should be null.
tocFlags	32-bit value	Flash boot parameters defined in Table 126 and Table 127 .
crc	32-bit value	

For a faster boot sequence, you can change the following two parameters:

- Boot clock frequency parameter “IMO/FLL clock frequency”
- Debug parameter “Wait Window Time”

Once in production, setting the wait window to 0 will speed up the overall boot time. These two parameters and others are part of the “tocFlags” variable in the `cy_stc_ps_toc_t` structure. [Table 126](#) and [Table 127](#) defines each of the parameters than can be set with the tocFlags element of the structure for the 1st generation and 2nd generation parts respectively.

Table 126 Flash boot options (tocFlags) for 1st generation parts

Parameter	Bits	Settings	Notes
IMO/FLL clock frequency	[1:0]	0 = 25 MHz (FLL) [Default] 1 = 8 MHz (IMO) 2 = 50 MHz (FFL) 3 = Reserved	CM0+ clock during boot. This clock will remain at this setting after Flash boot execution until the OEM firmware changes it.

(table continues...)

~~DRAFT~~
5 PSoC™ 6 application notes

Table 126 (continued) Flash boot options (tocFlags) for 1st generation parts

Parameter	Bits	Settings	Notes
Wait window time	[4:2]	0 = 20 ms 1 = 10 ms 2 = 1 ms 3 = 0 ms (No wait window) 4 = 100 ms 5-7 = Reserved	Determines the wait window to allow sufficient time to acquire the debug port.
Reserved	[30:5]		Not used
VALIDATE_APP_NORMAL	[31]	0 = No authentication 1 = Authentication	Setting this bit to 1 enables the authentication of the user code. The TOC2 must be complete and the public key must be written in to Sflash.

Table 127 Flash boot options (tocFlags) for 2nd generation parts

Parameter	Bits	Settings	Notes
IMO/FLL clock frequency	[1:0]	0 = 8 MHz, IMO, no FLL 1 = 25 MHz IMO + FLL 2 = 50 MHz IMO + FLL 3 = Use ROM boot clocks configuration (100 MHz)	CM0+ clock during boot. This clock will remain at this setting after Flash boot execution until the OEM firmware changes it.
Wait window time	[4:0]	0 = 20 ms 1 = 10 ms 2 = 1 ms 3 = 0 ms (No wait window) 4 = 100 ms 5-7 = Reserved	Determines the wait window to allow sufficient time to acquire the debug port.
SWJ (debug) pin state	[6:5]	0 = Do not enable SWJ pins 1 = Do not enable SWJ pins 2 = Enable SWJ pins 3 = Do not enable SWJ pins	Determines whether SWJ pins are configured in SWJ mode by Flash boot in SECURE LCS. <i>Note:</i> <i>SWJ pins may be enabled later in the user code.</i>

(table continues...)

~~5 PSoC™ 6 application notes~~~~DRAFT~~
Table 127 (continued) Flash boot options (tocFlags) for 2nd generation parts

Parameter	Bits	Settings	Notes
App authenticate disable	[8:7]	0 = Authentication is enabled 1 = Authentication is disabled 2 = Authentication is enabled 3 = Authentication is enabled	Determines whether the application image digital signature verification (authentication) is performed.

~~5 PSoC™ 6 application notes~~

Place the following code snippet in your CM0+ code (`main.c`) to generate the TOC2. There is also a redundant copy of the TOC2 that is an exact copy, RTOC2. You may need to make changes to this section of code to match your application.

```

/* Flashboot parameters stored in TOC2 for PSOC6ABLE2 devices */
#ifndef CY_DEVICE_PSOC6ABLE2
#define CY_PS_FLASHBOOT_FLAGS ((CY_PS_FLASHBOOT_VALIDATE_YES <<
CY_PS_TOC_FLAGS_APP_VERIFY_POS) \
| (CY_PS_FLASHBOOT_WAIT_20MS << CY_PS_TOC_FLAGS_DELAY_POS) \
| (CY_PS_FLASHBOOT_CLK_25MHZ << CY_PS_TOC_FLAGS_CLOCKS_POS))
#endif

/* Flashboot parameters stored in TOC2 for PSOC6A2M / PSOC6A512K devices */
#ifndef CY_DEVICE_PSOC6A2M || defined(CY_DEVICE_PSOC6A512K)
#define CY_PS_FLASHBOOT_FLAGS ((CY_PS_FLASHBOOT_VALIDATE_YES <<
CY_PS_TOC_FLAGS_APP_VERIFY_POS) \
| (CY_PS_FLASHBOOT_WAIT_20MS << CY_PS_TOC_FLAGS_DELAY_POS) \
| (CY_PS_FLASHBOOT_CLK_25MHZ << CY_PS_TOC_FLAGS_CLOCKS_POS) \
| (CY_PS_FLASHBOOT_SWJ_PINS_ENABLE << CY_PS_TOC_FLAGS_SWJ_ENABLE_POS))
#endif

/* TOC2 in SFlash */
CY_SECTION(".cy_toc_part2") __USED static const cy_stc_ps_toc_t cy_toc2 =
{
    .objSize      = sizeof(cy_stc_ps_toc_t) - sizeof(uint32_t), /* Object Size (Bytes)
excluding CRC */
    .magicNum     = CY_PS_TOC2_MAGICNUMBER,                         /* TOC2 ID (magic number) */
    .userKeyAddr  = (uint32_t)&CySecureKeyStorage,                 /* User key storage address */
    .smifCfgAddr  = 0UL,                                         /* SMIF config list pointer */
    .appAddr1     = CY_START_OF_FLASH,                            /* App1 (MCUBoot) start address
*/
    .appFormat1   = CY_PS_APP_FORMAT_CYPRESS,                      /* App1 Format */
    .appAddr2     = 0,                                            /* App2 (User App) start
address */
    .appFormat2   = 0,                                            /* App2 Format */
    .shashObj     = 1UL,                                         /* Include public key in the
SECURE HASH */
    .sigKeyAddr   = (uint32_t)&SFLASH->PUBLIC_KEY,                /* Address of signature
verification key */
    .tocFlags     = CY_PS_FLASHBOOT_FLAGS,                          /* Flash boot flags stored in
TOC2 */
    .crc          = 0UL,                                         /* CRC populated by
cymcuelftool at build time */
};

/* RTOC2 in SFlash, this is a duplicate of TOC2 for redundancy */
CY_SECTION(".cy_rtoc_part2") __USED static const cy_stc_ps_toc_t cy_rtoc2 =
{
    .objSize      = sizeof(cy_stc_ps_toc_t) - sizeof(uint32_t), /* Object Size (Bytes)
excluding CRC */
    .magicNum     = CY_PS_TOC2_MAGICNUMBER,                         /* TOC2 ID (magic number) */
    .userKeyAddr  = (uint32_t)&CySecureKeyStorage,                 /* User key storage address */
    .smifCfgAddr  = 0UL,                                         /* SMIF config list pointer */
}

```

5 PSoC™ 6 application notes

```

DRAFT

    .appAddr1      = CY_START_OF_FLASH,                      /* App1 (MCUBoot) start address
*/
    .appFormat1   = CY_PS_APP_FORMAT_CYPRESS,                /* App1 Format */
    .appAddr2      = 0,                                      /* App2 (User App) start
address */
    .appFormat2   = 0,                                      /* App2 Format */
    .shashObj     = 1UL,                                     /* Include public key in the
SECURE HASH */
    .sigKeyAddr   = (uint32_t)&SFLASH->PUBLIC_KEY,          /* Address of signature
verification key */
    .tocFlags     = CY_PS_FLASHBOOT_FLAGS,                  /* Flash boot flags stored in
TOC2 */
    .crc         = 0UL,                                     /* CRC populated by
cymcuelftool at build time */
};


```

5.15.4.2 CYPRESS™ standard application format

The initial project that is executed after Flash boot must be validated with a public RSA crypto key. To do this, the application must use the CYPRESS™ standard application format that includes a digital signature. This allows Flash boot to perform the validation during the boot process before the application is executed. The application format encapsulates the application binary, application metadata, and an encrypted digital signature. The user application includes both the CM0+ and CM4, see [Figure 552](#).

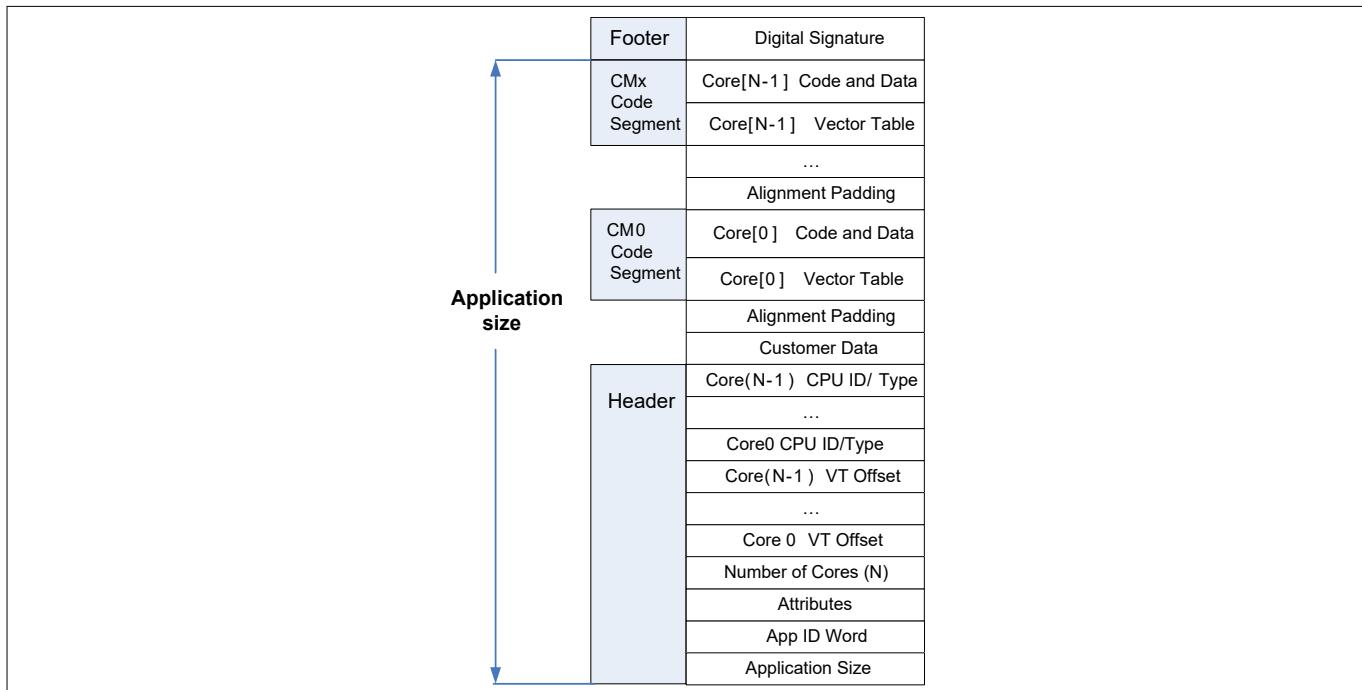


Figure 552 CYPRESS™ standard application format

[Table 128](#) provides the details of the header section. It defines the total size, the number of cores, the type of application, and the offset to each core application vector table.

Table 128 Application header details

Offset	Size	Item	Description
0x0	4 bytes	Application size	Flash image size in bytes

(table continues...)

5 PSoC™ 6 application notes

Table 128 (continued) Application header details

Offset	Size	Item	Description
0x4	4 bytes	Application ID word	<p>Identifies the type of the flash image:</p> <p>Bit 31 – 28: Always 0</p> <p>Bit 27 – 24: Major version (User defined)</p> <p>Bit 23 – 16: Minor version (User defined)</p> <p>Bit 15 – 0: Application ID. For example:</p> <ul style="list-style-type: none"> 0x0000 – 07FFF User Application ID 0x8001 – Flash boot 0x8002 – Security Image (NA) 0x8003 – Bootloader Values between 0x8004 – 0xFFFF Reserved
0x8	4 bytes	Attribute	Reserved for future use
0xC	4 bytes	Number of cores (N)	Number of cores used by the application
0x10 + (4*i)	4 bytes	Core(i) VT offset	Offset to vector table in Core(i) code segment
0x10 + (4*N) + (4*i)	4 bytes	Core(i) CPU ID/type	<p>Customer-assigned CPU ID and core index:</p> <p>Bit 31 – 20: CPU ID. This is the part number value from the CPUID [15:4] register in an Arm® device. (See below)</p> <p>Bit 7 – 0: Core index</p> <p>The core index is used to distinguish between multiple cores of the same type. For example, consider a system consisting of M0+ and two M4s. The M0+ is identified by CPUID=0xC60 and Core Index=0. The first M4 is identified by CPUID=0xC24 and Core Index=0. The second M4 is identified by CPUID=0xC24 and Core Index=1.</p>

To generate proper values for the application header, an instance of the `cy_stc_appheader_t` structure must be included in the project. An example of this structure can be seen in the code example at `proj_btldr_cm0p/`

5 PSoC™ 6 application notes

main.c source file. Although the header supports images for multiple CPU images, the bootloader in the example only includes an image for the CM0+.

```

DRAFT
*****
 * Application header and signature
 ****
#define CY_PS_VT_OFFSET      ((uint32_t)(&__Vectors[0]) - CY_START_OF_FLASH \
    - offsetof(cy_stc_ps_appheader_t, core0Vt)) /* CM0+ VT Offset */
#define CY_PS_CPUID          (0xC6000000UL)           /* CM0+ ARM CPUID[15:4] Reg shifted to
[31:20] */
#define CY_PS_CORE_IDX        (0UL)                  /* Index ID of the CM0+ core */

/** Secure Application header */
CY_SECTION(".cy_app_header") __USED
static const cy_stc_ps_appheader_t cy_ps_appHeader = {
    .objSize      = CY_BOOT_BOOTLOADER_SIZE - CY_PS_SECURE_DIGSIG_SIZE,
    .appId        = (CY_PS_APP_VERSION | CY_PS_APP_ID_SECUREIMG),
    .appAttributes = 0UL,                                /* Reserved */
    .numCores     = 1UL,                                /* Only CM0+ */
    .core0Vt      = CY_PS_VT_OFFSET,                   /* CM0+ VT offset */
    .core0Id      = CY_PS_CPUID | CY_PS_CORE_IDX, /* CM0+ core ID */
};

/* Secure Digital signature (Populated by cymcuelftool) */
CY_SECTION(".cy_app_signature") __USED CY_ALIGN(4)
static const uint8_t cy_ps_appSignature[CY_PS_SECURE_DIGSIG_SIZE] = {0u};

```

You should update the MAJOR and MINOR version constants in the *cy_ps_config.h* file (see code example) to the required value. The CY_USERAPP_ID is up to you; it should be between 0x0000 and 0xFFFF.

To generate the digital signature in the code, use the following. It will place the signature in the last 256 bytes of the user code area. For a complete example of creating and adding the public key, see [Appendix B - Creating crypto key pairs](#).

```

/* Secure digital signature (Populated by cymcuelftool) */
CY_SECTION(".cy_app_signature") __USED CY_ALIGN(4)
static const uint8_t cy_ps_appSignature[CY_PS_SECURE_DIGSIG_SIZE] = {0u};

```

5.15.4.3 Infineon secured boot RSA public key format

The SFlash region stores the public key. It is stored in a binary format, not the ASCII format generated by OpenSSL. The modulus, exponent, and three coefficients are pre-calculated to speed up the validation. [Figure 553](#) shows the format.

5 PSoC™ 6 application notes

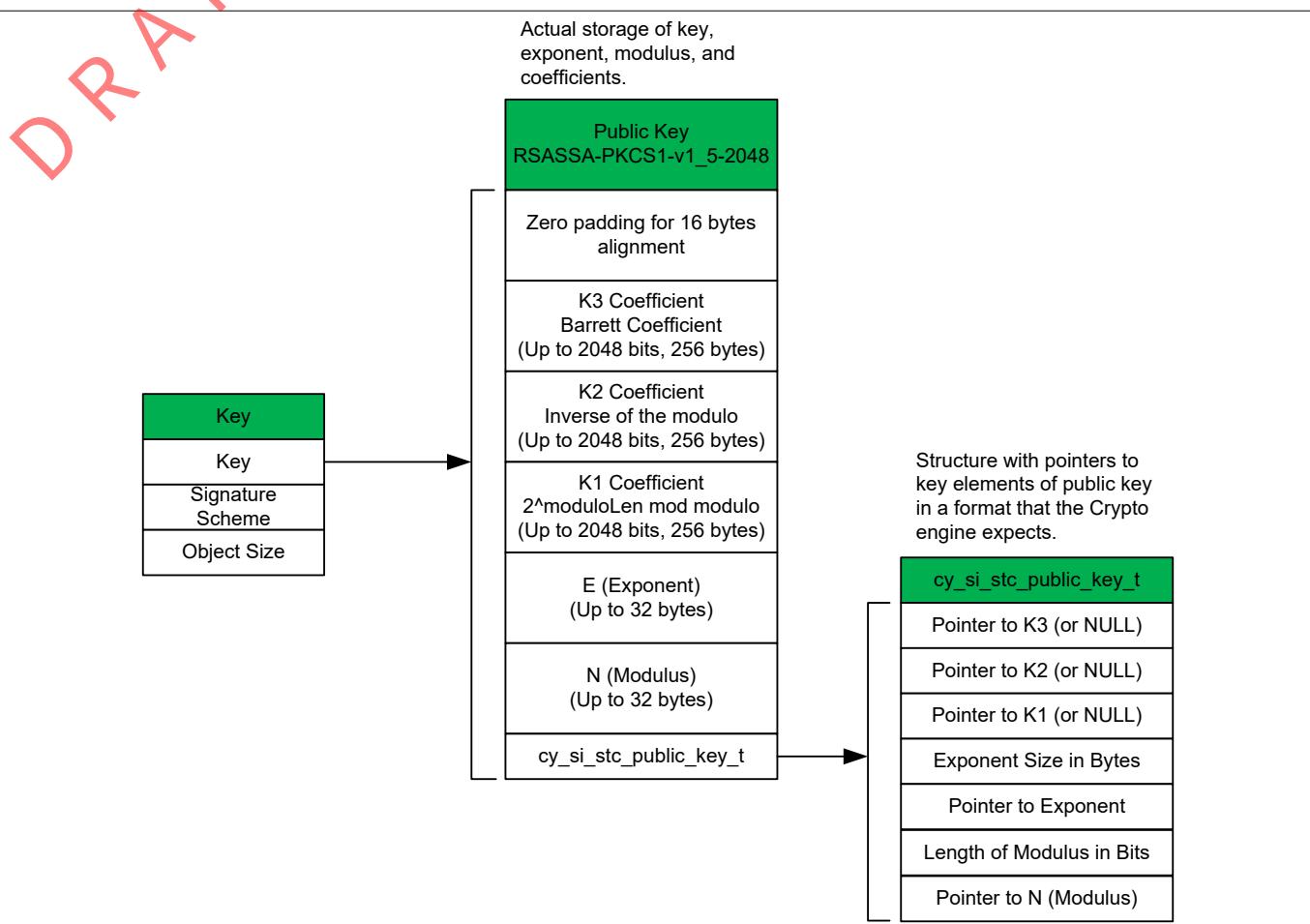


Figure 553 Crypto key structures

The key is stored in three structures:

1. The first structure “Key” is stored as an object that can easily be included in the Secure_HASH calculation. The “Signature Scheme” defines the structure of the key.
 - This example uses RSASA-PKCS1-v1_5-2048. The “Object Size” contains the full size of the public key object, which contains the entire three structures.
2. The second structure contains the individual pieces of the public key: coefficients (K1, K2, K3), exponent (E), and modulus (N). These values must be stored in a little-endian list of bytes. OpenSSL generates these values in a big-endian format.
3. The third structure is a list of pointers to each piece of the public key, which is the format required for a call to the Crypto driver. The key is stored in this expanded to speed up the code verification process.

For the companion code example (CE234992), there is already a default key generated. This will work fine for development, but you should not use this for production. To generate a new custom key pair, see [Appendix B - Creating crypto key pairs, section 8.1](#).

5 PSoC™ 6 application notes

5.15.4.4 Programming eFuse to change the lifecycle stage

Attention: This is the most critical section to perform properly. The eFuse bits can be programmed only from 0 to 1. If you program these bits incorrectly, you will need to remove the device from the board and dispose of it. You should not proceed to this step until you have proven all other code is working properly. Also, the only way to update the code after entering the SECURE state and disabling the debug ports is to have a proven bootloader working. The supply voltage on pin V_{DDIO0} must be set to 2.5 V to program eFuse. You can also power the entire device with 2.5 V if that is more convenient.

Code that is used to generate eFuse data can be found in companion code example (CE234992) at /proj_btldr_cm0p/source/cy_ps_efuse.c. This file contains the structures that compile and create the eFuse data. By default, no eFuse data will be generated, just in case the project is compiled and programmed by accident. To enable eFuse programming, a line in cy_ps_efuse.h must be changed from:

```
“#define CY_EFUSE_AVAILABLE (0)”
```

to

```
“#define CY_EFUSE_AVAILABLE (1)”.
```

A byte of data is required to program each bit of the eFuse. The following pattern is used to program, validate, or set as ‘don’t care’ each bit of eFuse.

```
/* EFUSE bit action macros */
#define CY_EFUSE_STATE_SET      (0x01U) /* Tell programmer to set the EFUSE bit */
#define CY_EFUSE_STATE_UNSET    (0x00U) /* Tell programmer to check that the EFUSE bit is not
set */
#define CY_EFUSE_STATE_IGNORE   (0xffU) /* Tell programmer to ignore the EFUSE bit */
```

Find the section shown below in the file cy_ps_efuse.c. This is where you change the lifecycle to either SECURE_WITH_DEBUG or SECURE, you can only set one of these bits. If one of these two bits is already set, the programmer will not program the other. To program the SECURE or SECURE_WITH_DEBUG bit, change the constant CY_EFUSE_STATE_IGNORE to CY_EFUSE_STATE_SET. The NORMAL bit will already be set from the factory and should remain as “CY_EFUSE_STATE_IGNORE”.

```
.LIFECYCLE_STAGE =
{
    CY_EFUSE_STATE_IGNORE,          /* NORMAL lifecycle already set - ignore */
    CY_EFUSE_STATE_IGNORE,          /* SECURE_WITH_DEBUG lifecycle */
    CY_EFUSE_STATE_IGNORE,          /* SECURE life cycle */
    CY_EFUSE_STATE_IGNORE,          /* Infineon use only - ignore */
    CY_EFUSE_LIFECYCLE_RESERVED0 /* Reserved bits ignored */
},
```

5 PSoC™ 6 application notes

5.15.4.4.1 Using CYPRESS™ Programmer

DRAFT
Important notes:

1. When using CYPRESS™ Programmer along with the Infineon MiniProg4 (CY8CKIT-005) to change the lifecycle stage to SECURE, it does more than just programming the eFuse bits. The following are the actual steps. Some programmers may not perform all three steps as outlined below.
 - Validates the SFlash area with the internal Factory_HASH to make sure the part has not been modified after leaving the factory. If an error is found, you should not advance to the SECURE lifecycles stage.
 - Generates Secure_HASH based on the TOC2 entries. By default, Secure_HASH includes all of SFlash. Additional areas from user flash may be included if additional entries are added in TOC2. This hash is written into the Secure_HASH area of the eFuse along with the number of zeros in the hash. This guarantees that the hash cannot be modified by simply changing zeros to ones.
 - Programs the eFuse bits for access restrictions and the lifecycle SECURE bit
2. Programming the EFuses is irreversible, and care should be taken to verify the settings before blowing them. Incorrect settings may brick the device permanently.
3. In SECURE lifecycle mode, if the secure access restrictions are set to enable the debug access ports, the GPIOs need to be configured by the user for the debugger to get access to the debug ports. This is demonstrated in the function *configure_swj* in file *proj_btldr_cm0p/source/main.c*. This function is disabled by default but can be enabled by setting the macro *CONFIGURE_SWJ_PINS* to 1.

5.15.4.4.2 Setting up CYPRESS™ Programmer

Use the following settings when programming the kit using CYPRESS™ Programmer.

- **Reset chip** should be checked
- **Program Security Data** should be un-checked (**Note:** When programming EFuse, this should be checked)
- **Voltage** set to 3.3V (**Note:** When programming EFuse, set this to 2.5V)
- **Reset Type** set to Soft
- **Sflash Restrictions** set to "Erase/Program USER/TOC/KEY allowed"

Specify the hex file to be programmed and then click **Connect** and **Program**.

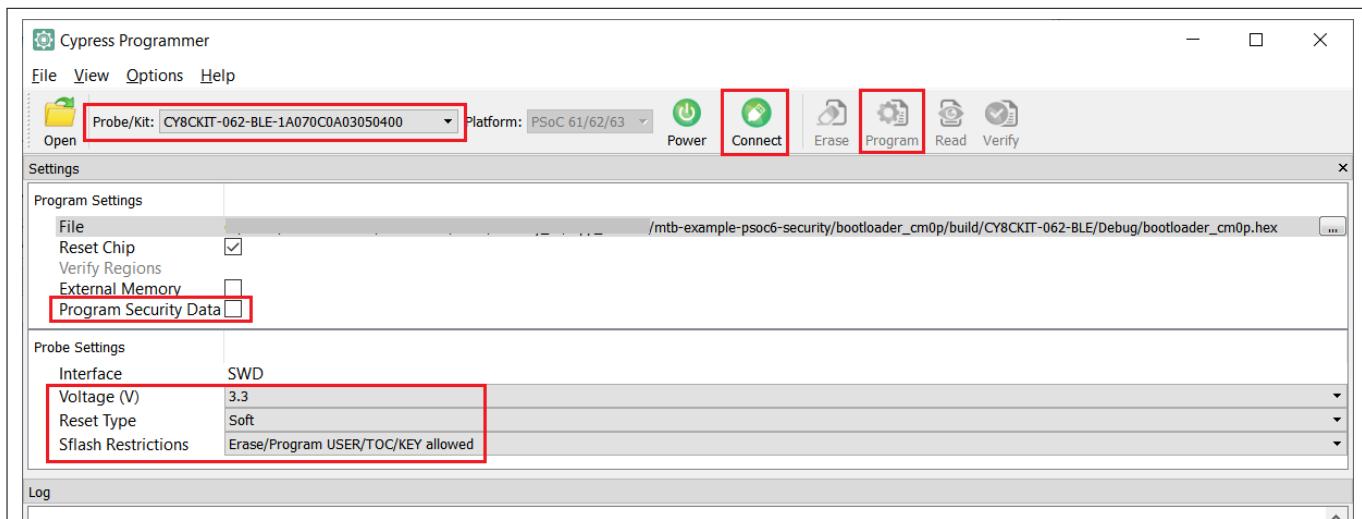


Figure 554 CYPRESS™ Programmer settings for EFuse programming

Note: For more specific information about the operation of the code example after the device is programmed, refer to the code example *ReadMe.md* file.

5 PSoC™ 6 application notes**5.15.5 Dual CPU design considerations**

~~DRAFT~~

Making use of the dual-CPU feature in the PSoC™ 62/63 adds yet another security option. Although the two CPUs share the same internal Flash and SRAM, the SMPUs allow you to isolate sections of memory as if each CPU had its own memory. The IPC block provides a convenient way to pass 32-bit words between the two CPUs. If large amounts of data need to be transferred between the two CPUs, a section of the SRAM can be configured to be accessible by both CPUs. This way you can pass a pointer to a structure that includes a large buffer of data.

One way to make use of the dual-CPU feature for added security is to divide your application between SECURE and NON-SECURE code. Run the SECURE code in CM0+ and the NON-SECURE part of the application on CM4. This way you can limit the SECURE code and more thoroughly test it. Also, you would run the more complicated software such as Wi-Fi or Bluetooth® stacks that are more difficult to test on CM4.

One example is a Wi-Fi enabled door lock with finger print sensor. The code that operates the SECURE part of the door lock such as the finger print sensor and the hardware mechanism would run on CM0+ and the Wi-Fi stack would run on CM4.

5 PSoC™ 6 application notes~~DO NOT USE~~
5.15.6 Summary

This application note has shown how the PSoC™ 6 MCU hardware can be used to create a secured system. Every application has different requirements and the flexibility of the PSoC™ security hardware can be used to adapt to those needs. As mentioned earlier, it is important to think about security from the beginning of your project and analyze each point of access to the code and data, do not let security be an afterthought. Determining up front what is required will reduce the chance that you will need to re-architect your project late in your project schedule.

[Appendix A - Code example of a security application template](#) in this document describes a code example that can be used as a template to implement what has been discussed in the application note.

~~5 PSoC™ 6 application notes~~

~~DRAFT~~ 5.15.7 Appendix A - Code example of a security application template

This appendix explains a code example that demonstrates most of the topics discussed in this application note. The code example [CE234992 PSoC™ 6 MCU: Security Template](#), includes all the source code, Makefiles, and scripts required to build, link, and sign the code. The following is a list of features demonstrated:

- Bootloader based on the industry standard MCUBoot (CM0+)
- Bootloader is cryptographically signed
- Dual-CPU operation; user applications for both CM0+ and CM4
- Supports device firmware update (DFU) with the standard UART interface
- FreeRTOS running on CM4
- CM0+ and CM4 application bundle signed
- Full Chain of Trust
- Isolated CPUs using SMPUs
- Communication between CM0+ and CM4

This code example consists of the following three projects:

1. proj_btldr_cm0p (MCUBoot bootloader CM0+)
2. proj_cm0p (CM0+ user project)
3. proj_cm4 (CM4 user project)

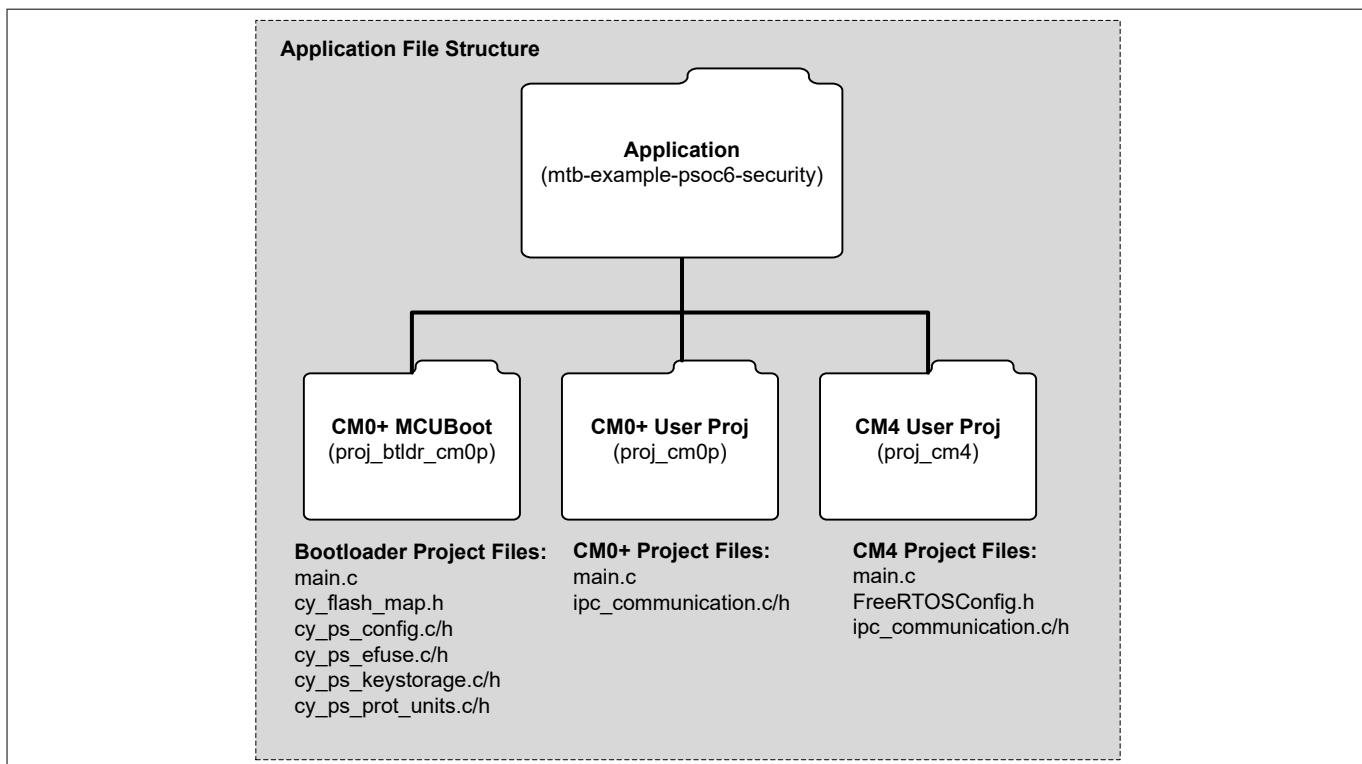


Figure 555

Application file structure

5 PSoC™ 6 application notes**5.15.7.1 ~~DRAFT~~ Bootup flow**

This flow assumes that the device has already been moved to the SECURE lifecycle stage.

1. CM0+ starts from reset.
2. CM0+ executes the internal ROM code that does the following:
 - Loads the trim values
 - Verifies that the second half of the boot code (Flash boot) and the user's public key are intact
3. The Flash boot code verifies that the user's bootloader (proj_btldr_cm0p) is signed by the owner of the public key (RSA-2048) and that the code has not been corrupted.
4. Jumps to the bootloader.

5 PSoC™ 6 application notes

DRAFT

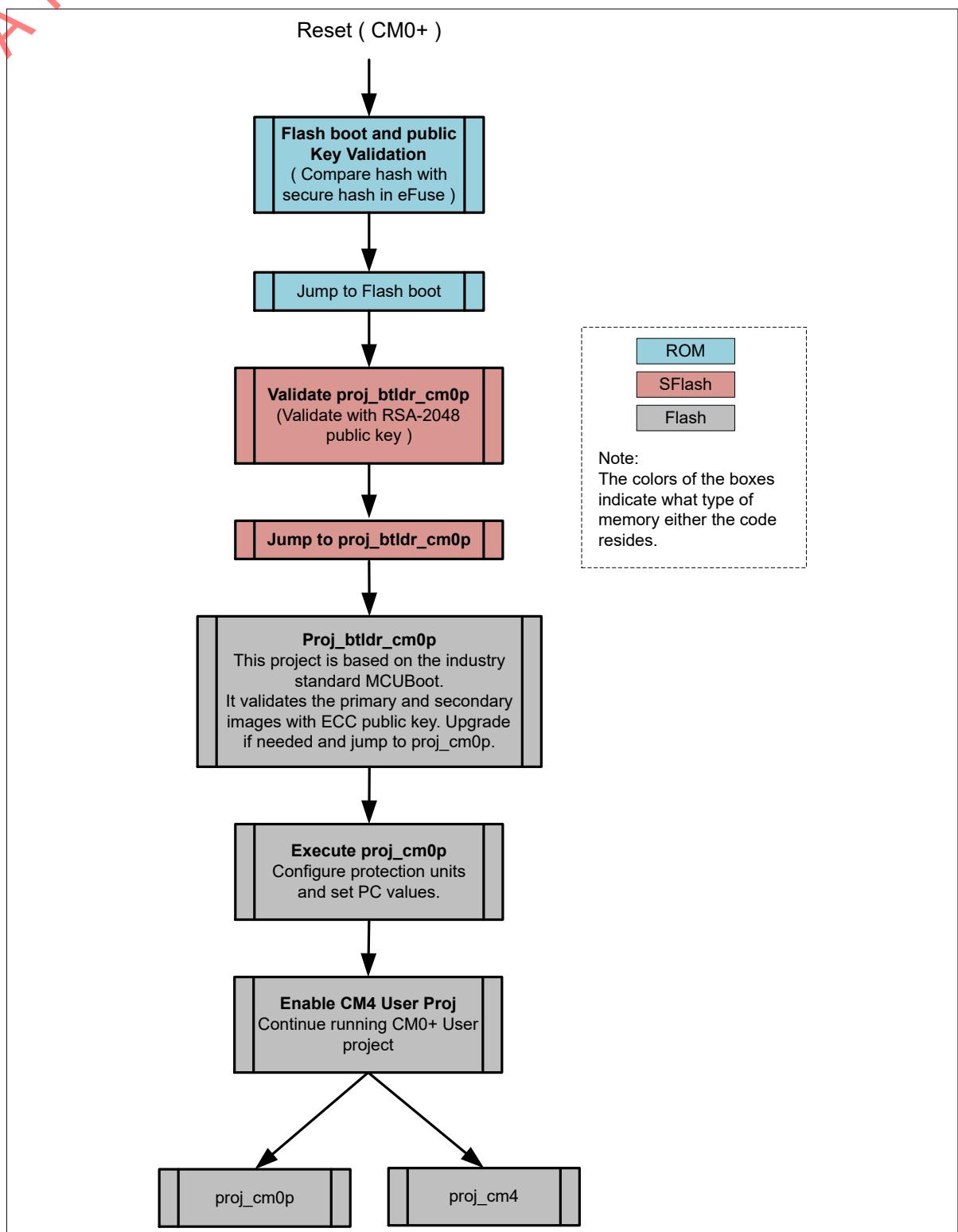


Figure 556 Application boot flow

5. The bootloader configures the protection units to isolate CM0+ and CM4 projects.
 - Review the code in the `/proj_btldr_cm0p/cy_ps_prot_units.c` file to learn how the SMPUs and Protection Context registers are set up.
6. The code checks to see if there is a newer version of the user application bundle, which includes both the CM0+ and CM4 applications. If a newer version is available, the code does the following:
 - a. Copies the new firmware image to the primary slot

5 PSoC™ 6 application notes

- DRAFT**
- b. Validates the new firmware image with the ECC key
 - c. If the image is verified, jumps to the user CM0+ project (proj_cm0p)
 7. The CM0+ application enables CM4 and continues to execute its application.
 - In the associated code examples that is used to demonstrate the firmware flow, the CM0+ application is nothing more than a LED blink application. You can remove these two lines of code and replace it with your own application.
 8. CM4 comes out of reset, and does the following:
 - a. Configures the required hardware
 - b. Initializes a few FreeRTOS tasks

One of these tasks is the DFU (device firmware upgrade), which listens on a serial port waiting for a command to start the download of a new version of the application bundle.

5.15.7.2 Application Chain of Trust (CoT)

The following diagram illustrates the Chain of Trust for this project. Note that there are two different types of crypto keys used to validate the full application. The PSoC™ 62/63 MCU natively uses an RSA-2048 key; the bootloader based on MCUBoot uses ECC by default.

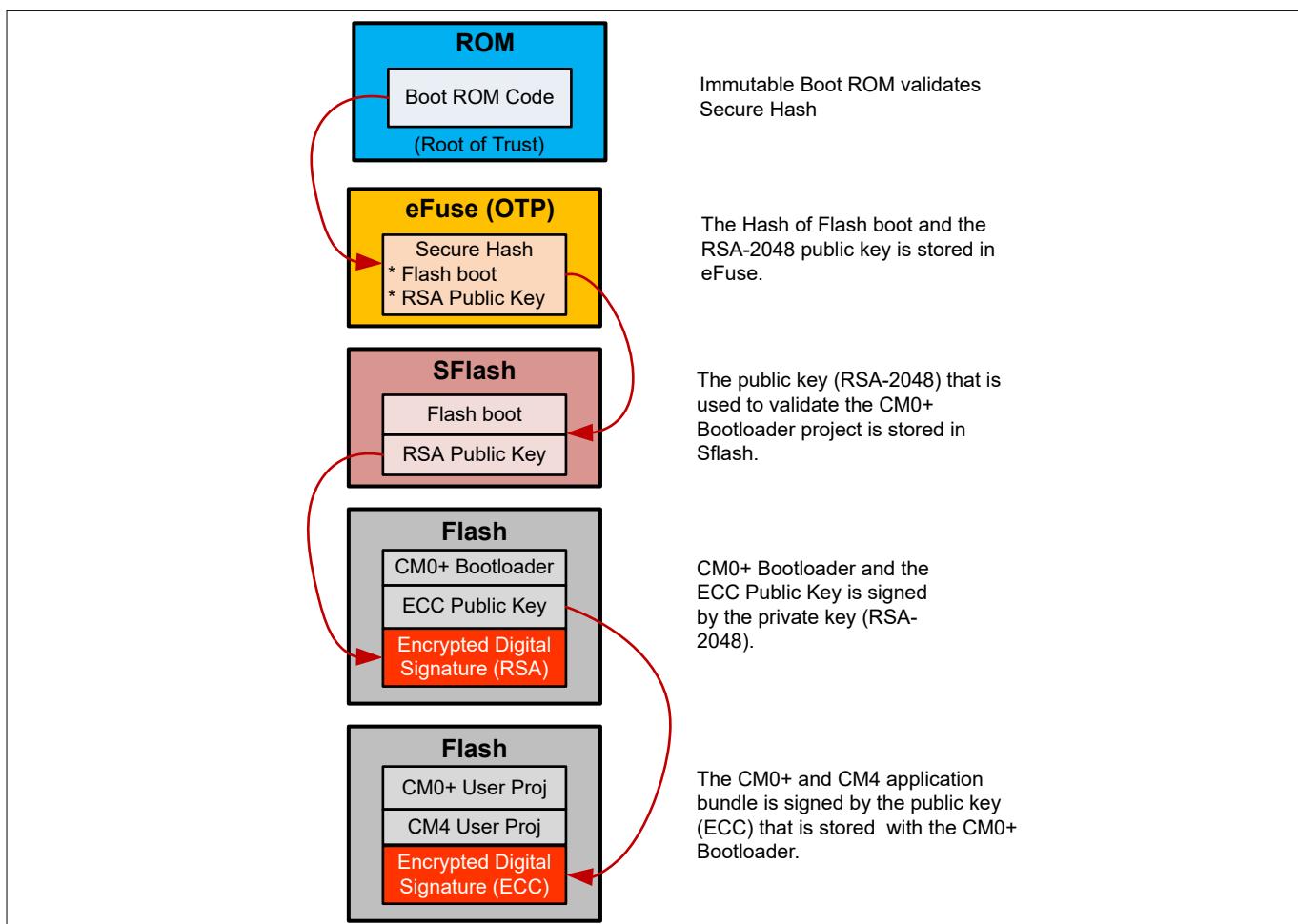


Figure 557

Chain of Trust

5 PSoC™ 6 application notes

~~DRAFT~~ 5.15.7.3 Project memory map

The memory map for the example project is shown below. The diagrams also show the access of the memory by the protection context.

The flash memory is broken up into four major areas:

- Bootloader
- Protected memory
- Primary slot
- Secondary slot

The bootloader area is not intended to be updated in the field and therefore must be thoroughly tested before release. Most of the code in the bootloader area is a port from MCUBoot, the industry standard bootloader infrastructure.

The protected memory region contains the sensitive data; only CM0+ has direct access to it. While some of this data could be preprogrammed from the factory, other parts could be used for dynamic storage.

The primary slot is the area in which the application code is executed. Both CM0+ and CM4 execute code in this region. Only CM0+ can write to this area while executing the bootloader; CM4 cannot write to this region.

The secondary slot is the area used to store an updated project. In most applications, it is up to CM4 to perform the operation because it is most likely to be the CPU communicating to the outside world via Wi-Fi, Bluetooth®, or DFU. If required, the CM0+ application code can update the code in the secondary slot.

Table 129 Protection context summary

Protection context (PC)	CPU access	Notes
0	CM0+	Default state of CM0+. It is used to configure the protection units at the beginning of the bootloader code. All memory and registers are accessible in PC=0; all system calls operate at PC=0 as well.
1	CM0+	The bootloader runs at Protection Context 1 after protection units have been configured. All areas that need to be accessed by the bootloader have PC=1 as part of their mask.
2	CM0+	Reserved for the protected memory that could be used for secure storage. CM0+ must switch to PC=2 while accessing the area.
4	CM4	Domain of the CM4 CPU. It must access its portion of the primary slot and all of the secondary flash area. The CM4 code is responsible for updating the code in the secondary slot.

5 PSoC™ 6 application notes

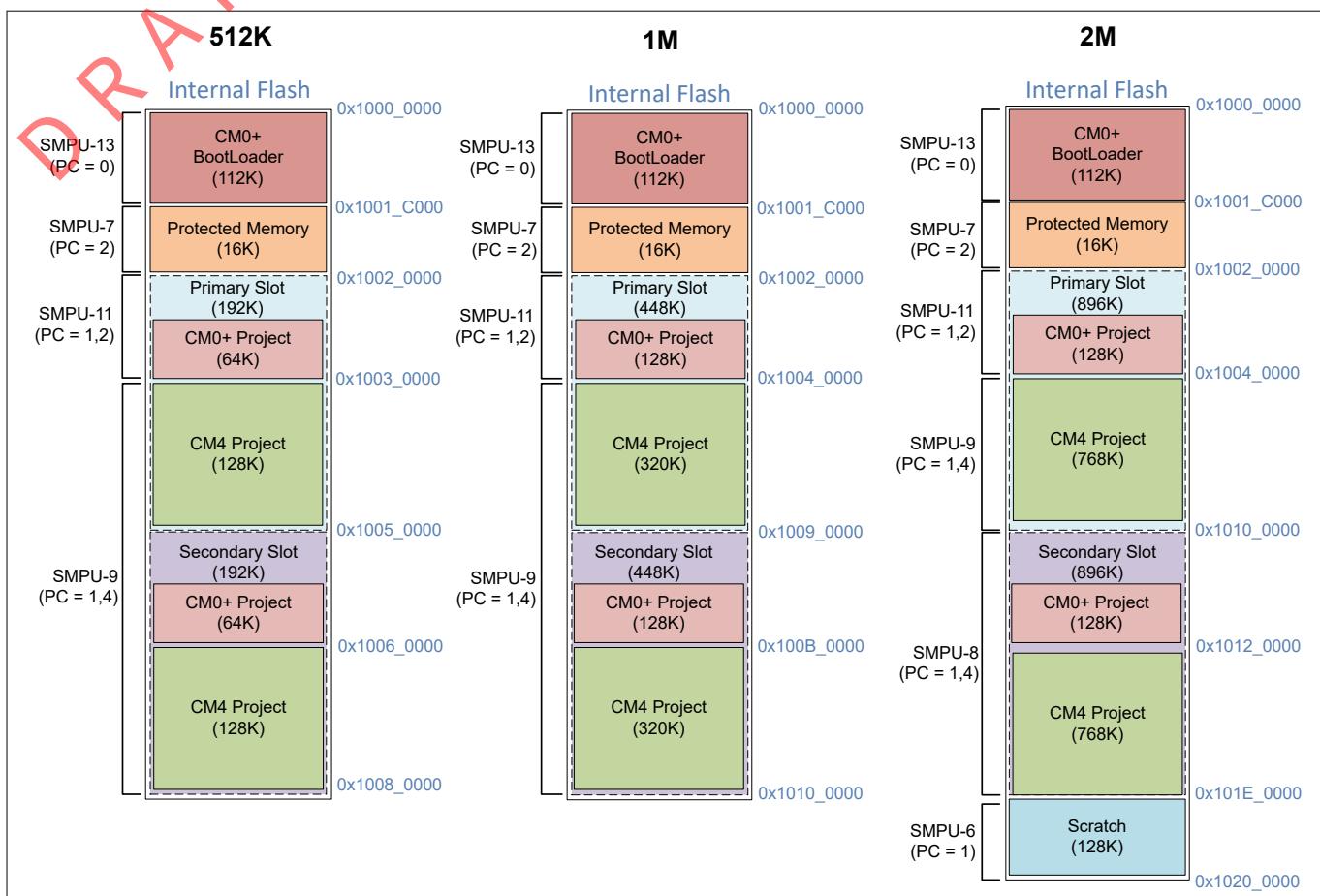


Figure 558 Project flash memory maps for 512K, 1M, and 2M flash parts

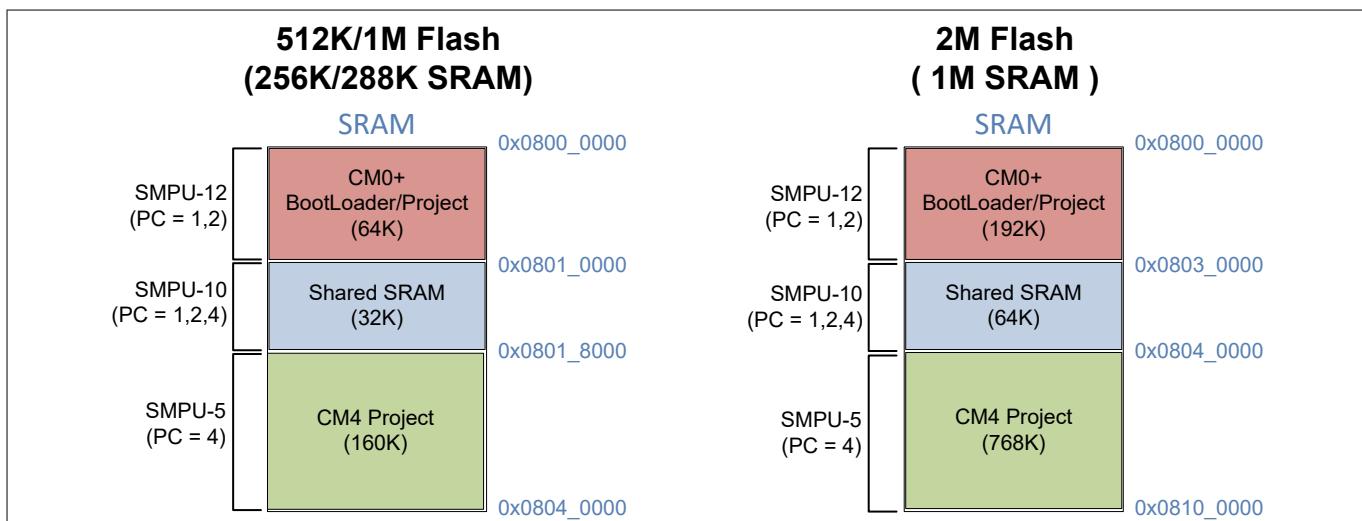


Figure 559 Project SRAM memory maps for 512K, 1M, and 2M flash parts

~~5 PSoC™ 6 application notes~~

~~5.15.8 Appendix B - Creating crypto key pairs~~

The ModusToolbox™ software installation includes the tools required to generate and format the private/public key pairs, including OpenSSL and Python 3.7+. To access these tools without updating any system paths, use the “Modus Shell” command-line shell.

The PSoC™ 62/63 Flash boot code requires an RSA-2048 public key to be stored in SFlash, and the first user code to be signed with the corresponding private key to boot in the SECURE lifecycle stage. In the companion code example (CE234992), “proj_btldr_cm0p” is the first user code.

The proj_btldr_cm0p project is a port of the open source MCUboot project. This project boots the actual user project code; it also requires a crypto key pair to sign and validate the user project. MCUboot uses the ECDSA algorithm instead of RSA, so it cannot use the same RSA key that is used by Flash boot. This appendix provides details of what is required to generate the keys for both RSA and ECC.

5.15.8.1 Generating the RSA key pair

The companion code example (CE234992) includes a default key that is already generated. This key is sufficient for development, but do not use this for production. These steps will generate a custom private/public key pair in the *keys* directory and generate a C-compatible public key, but you must manually copy the generated C code to the *cy_ps_keystorage.c* file.

Do the following to generate a new custom key pair:

1. Start the “modus-shell” application that is installed along with ModusToolbox™ software.
2. Navigate to *mtb-psoc6-example-security/proj_btldr_cm0p* directory.
3. Execute the command **make rsa_keygen**.
 - This command creates a file in the *keys* subdirectory called *rsa_to_c_generated.txt*.
4. Copy the following arrays from the *rsa_to_c_generated.txt* file and replace those in *cy_ps_keystorage.c*.
 - *.moduleData[]*
 - *.expData[]*
 - *.barrettData[]*
 - *.inverseModuleData[]*
 - *.rBarData[]*

5.15.8.2 Generating the ECC key pair

Generating the ECC key pair required for the bootloader (MCUboot) is similar to the RSA key pair generation but simpler.

1. Start the modus-shell application.
2. Navigate to the *mtb-psoc6-example-security/bootloader_cm0p* directory.
3. Execute the command **make ecc_keygen**.

This will create the following files in the *keys* folder:

- *cypress-test-ec-p256.pem* (private key)
- *cypress-test-ec-p256.pub* (public key in a C-like array)
- *ecc-public-key-p256.h* (public-key header file in C-like array)

The public key will automatically be incorporated into the MCUboot build, you do not need to edit the files.

~~5 PSoC™ 6 application notes~~

~~5.15.8.2.1 Example RSA private/public key files~~

The “make rsa_keygen” command creates the following two other files in the *keys* directory:

- *rsa_private_generated.txt*
- *rsa_public_generated.txt*

These files are generated by OpenSSL and are used to create the *rsa_to_c_generated.txt* file.

Note: These listings show examples of what the files will look like. The actual data in the files may differ from what is shown here.

The *rsa_private_generated.txt* private key file will look like as follows:

```
-----BEGIN RSA PRIVATE KEY-----
MIIEowIBAAKCAQEAt8ygBxZNdFUHz7m7sQXPYinpAui05aFoEJsiopNTScohuCGA
BTtReTI7V5x8h/+etDrnWG+AhYNHKT+uh705ZFZU7MBd/69n9jBWFJDfbJhXfvv69
r7uPwVJK705GLkUnVNCxHiz6/CGCYs6RyWQ9xQxMuSqCCjKOxhk5x0SwjMeAh3Pm
MR+AZFd2W7fkADjFDODgb9mDGp+49z0+v/nGTgNR5PEbMbBwjAjcNyxeM5LrtzTz
8pongkeg/XBBsaDaZPpqHm67HknMAHUeTKACx9vLHHksjm8w6n55/QGwiGBUd1fk
z0gQii4rGluKoCft76sqdpdCQuahF2EACTb0MwIDAQABoIBAF6UUZTUCTA10rs1
3DuPvdPJtTn16gKIOECzU/NM1IMYHJnfwzzt9VLKy10HDZ35+XemcWMOp5H1k+h
9UUsWzFyhtJPg51Um+Pc1/bzk2e2/AwrfOMFMFqU10pbjvJIiAm872Pb+fmZm3p
1mNHzfFkDucJ1Ljio02VFYJQ+ni2IIJGi/QxRBOS0I5PAIUQCuLoiRx3m061uJC5
DaZGZLm0/tpz1FNGNMHuP+iQH5UlWIRhqSnEH0dOUWikUhZU2d3MwEjqS0I/tx12
nsUqsxooG7FNrtvo2QzSX0mYXJml0EuFJpwSrMsQnlq0uK13SBXkfhoCoj0wNrj
sZetAOECgYEAE7ElrvtqC6fTQFiN49L3W+/WdMLMtPpguUhWGZB6A+/HaTM300xMj
Ks9JXdeA+LMw9k3Q9ukAbisy/PphCSHiHphbhzPzFQvluxXswh/kh82X+TFEs+jt
9ceAuA45cULZHPXSqljmHet4wXeiDH1tCufG6VP3JL90hGkRqWZwJ30CgYEAxyp
qKNsdVzdAHFVcX2Wd7kUSbs2TrmIos1CmyjdfEtPbWjzWbRCFzAjpqVDjNA67ZeH
DBo68xjaNv0unkBj/Sc1FQxprPXbPc0T6W4n6J2w0C1jabnCiXamHKoaNenbze5+
K5n1LyuLnnyjWt8p63JvH1cUU61djq+M+VTn0W8CgYEak7NmhANBmpfc+uokNQT1
gMDVC1wIngvdRuFz900GXPiYH+k/sWBB8Nc/3C5bUFBC8osKCAUJ8uT7bltrnbz
mGLxxX3pqq+sZxxE7jtX+FpcduoJJFwrX6rgWrBx0s/SwZsgn2RCQFQ0QubShPJo
mBe8L6RRbkS3BmpVI00qM1UCgYBfssstv4bfIXatE/zDTrEmV1jYNh0z1v3nM4V1U
ObrUc0vjgz8Mp/XqNpOrDFb9X0Ix0VJvm1shXzveNBJ/6QkjLyfuxjyXhDyp4dXj
oa9DtCK2pljTARReAcmlISry4h32FaA1R7wH7ijm7jTfdEi9oY3RRzbaL9ARVM0
rHu/uQKBgB7cZeXHu2zPhKN91uuB5NI8N3SoGhwajj0DM8Nb4MQdWF16BHvgb10Z
Ld12+nuRnzim7rpS8vN1VRS2KnmTvdGEb8yw5hMxGN3J8yKg8Gs1bMaJ4rmlh7j
GBZrx4ttq9fjipfgTYqN1QTWabxRqG8S0LTIjWpiVZMZOGoFSOLt
-----END RSA PRIVATE KEY-----
```

The “*rsa_public_generated.txt*” public key file will look like as follows:

```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEAAQ8AMIIIBCgKCAQEAt8ygBxZNdFUHz7m7sQXP
YinpAui05aFoEJsiopNTScohuCGABTtReTI7V5x8h/+etDrnWG+AhYNHKT+uh705Z
FZU7MBd/69n9jBWFJDfbJhXfvv69r7uPwVJK705GLkUnVNCxHiz6/CGCYs6RyWQ9
xQxMuSqCCjKOxhk5x0SwjMeAh3PmMR+AZFd2W7fkADjFDODgb9mDGp+49z0+v/nG
TgNR5PEbMbBwjAjcNyxeM5LrtzTz8pongkeg/XBBsaDaZPpqHm67HknMAHUeTKAC
x9vLHHksjm8w6n55/QGwiGBUd1fkz0gQii4rGluKoCft76sqdpdCQuahF2EACTb0
MwIDAQAB
-----END PUBLIC KEY-----
```

5 PSoC™ 6 application notes**5.15.8.3 ~~DRAFT~~ Editing the RSA key C file (cy_ps_keystorage.c)**

The following is an example of the `cy_ps_keystorage.c` file. The sections shown in *italics* are areas that need to be replaced with the contents of the `rsa_to_c_generated.txt` file after running the `make rsa_keygen` command.

The final step to updating the public key in the secure image is to copy the code in the generated `rsa_to_c_generated.txt` file to the `cy_keystorage.c` source file, which is part of the secure image project.

5 PSoC™ 6 application notes

An example of this file after being updated is shown below. The replacement code from the generated key data is shown below in *italics*.

```

DRAFT

/*****\file cy_ps_keystorage.c
* \version 1.20
*
* \brief
* Secure key storage for application usage.
*
*****\copyright
* Copyright 2017-2018, Cypress Semiconductor Corporation. All rights reserved.
* You may use this file only in accordance with the license, terms, conditions,
* disclaimers, and limitations in the end user license agreement accompanying
* the software package with which this file was provided.
*****/

#include <source/cy_ps_keystorage.h>

#if defined(__cplusplus)
extern "C" {
#endif

/* Secure Key Storage (Note: Ensure that the alignment matches the Protection unit
configuration) */
CY_ALIGN(1024) __USED const uint8_t CySecureKeyStorage[CY_PS_SECURE_KEY_ARRAY_SIZE]
[CY_PS_SECURE_KEY_LENGTH] = {
    {0x00u}, /* Insert user key #1 values */
    {0x00u}, /* Insert user key #2 values */
    {0x00u}, /* Insert user key #3 values */
    {0x00u} /* Insert user key #4 values */
};

/* Public key in SFlash */
CY_SECTION(".cy_sflash_public_key") __USED const cy_ps_stc_public_key_t cy_publicKey =
{
    .objSize = sizeof(cy_ps_stc_public_key_t),
    .signatureScheme = CY_PS_PUBLIC_KEY_RSA_2048,
    .publicKeyStruct =
    {
        .moduloAddr      = (uint32_t)&(SFLASH->PUBLIC_KEY) +
offsetof(cy_ps_stc_public_key_t, moduloData),
        .moduloSize      = CY_PS_PUBLIC_KEY_SIZEOF_BYTE * CY_PS_PUBLIC_KEY_MODULOLENGTH,
        .expAddr         = (uint32_t)&(SFLASH->PUBLIC_KEY) +
offsetof(cy_ps_stc_public_key_t, expData),
        .expSize         = CY_PS_PUBLIC_KEY_SIZEOF_BYTE * CY_PS_PUBLIC_KEY_EXPLENGTH,
        .barrettAddr     = (uint32_t)&(SFLASH->PUBLIC_KEY) +
offsetof(cy_ps_stc_public_key_t, barrettData),
        .inverseModuloAddr = (uint32_t)&(SFLASH->PUBLIC_KEY) +
offsetof(cy_ps_stc_public_key_t, inverseModuloData),
        .rBarAddr        = (uint32_t)&(SFLASH->PUBLIC_KEY) +
offsetof(cy_ps_stc_public_key_t, rBarData),
    }
};

```

5 PSoC™ 6 application notes

DRAFT

```

},
.moduloData =
{
    0x33u, 0xF4u, 0x36u, 0x09u, 0x00u, 0x61u, 0x17u, 0xA1u,
    0xE6u, 0x42u, 0x42u, 0x97u, 0x76u, 0x2Au, 0xABu, 0xEFu,
    0xD3u, 0x27u, 0xA0u, 0x8Au, 0x5Bu, 0x1Au, 0x2Bu, 0x2Eu,
    0x8Au, 0x10u, 0xE8u, 0xCCu, 0xE4u, 0x57u, 0x77u, 0x54u,
    0x60u, 0x88u, 0xB0u, 0x01u, 0xFDu, 0x79u, 0x7Eu, 0xEAu,
    0x30u, 0x6Fu, 0x8Eu, 0x2Cu, 0x79u, 0x1Cu, 0xCBu, 0xDBu,
    0xC7u, 0x02u, 0xA0u, 0x4Cu, 0x1Eu, 0x75u, 0x00u, 0xCCu,
    0x49u, 0x1Eu, 0xBBu, 0x6Eu, 0x1Eu, 0x6Au, 0xFAu, 0x64u,
    0xDAu, 0xA0u, 0xB1u, 0x41u, 0x70u, 0xFDu, 0xA0u, 0x47u,
    0x82u, 0x27u, 0x9Au, 0xF2u, 0xF3u, 0x34u, 0xB7u, 0xEBu,
    0x92u, 0x33u, 0x5Eu, 0x2Cu, 0x37u, 0xDCu, 0x08u, 0x8Cu,
    0x70u, 0xB0u, 0x31u, 0x1Bu, 0xF1u, 0xE4u, 0x51u, 0x03u,
    0x4Eu, 0xC6u, 0xF9u, 0xBFu, 0xBEu, 0x33u, 0xF7u, 0xB8u,
    0x9Fu, 0x1Au, 0x83u, 0xD9u, 0x6Fu, 0xE0u, 0xE0u, 0x0Cu,
    0xC5u, 0x38u, 0x00u, 0xE4u, 0xB7u, 0x5Bu, 0x76u, 0x57u,
    0x64u, 0x80u, 0x1Fu, 0x31u, 0xE6u, 0x73u, 0x87u, 0x80u,
    0xC7u, 0x8Cu, 0xB0u, 0x44u, 0xC7u, 0x39u, 0x19u, 0xC6u,
    0x8Eu, 0x32u, 0x0Au, 0x82u, 0x2Au, 0xB9u, 0x4Cu, 0x0Cu,
    0xC5u, 0x3Du, 0x64u, 0xC9u, 0x91u, 0xCEu, 0x62u, 0x82u,
    0x21u, 0xFCu, 0xFAu, 0x2Cu, 0x1Eu, 0xB1u, 0xD0u, 0x54u,
    0x27u, 0x45u, 0x2Eu, 0x46u, 0x4Eu, 0xEFu, 0x4Au, 0x52u,
    0xC1u, 0x8Fu, 0xBBu, 0xAFu, 0xBDu, 0xFEu, 0xBEu, 0xDFu,
    0x15u, 0x26u, 0xDBu, 0x37u, 0x24u, 0x85u, 0x15u, 0x8Cu,
    0xFDu, 0xD9u, 0xEBu, 0x7Fu, 0x17u, 0x30u, 0x3Bu, 0x95u,
    0x15u, 0x59u, 0xEEu, 0xECu, 0xA1u, 0xEBu, 0x4Fu, 0xCAu,
    0xD1u, 0x60u, 0x21u, 0xE0u, 0x1Bu, 0xD6u, 0xB9u, 0x0Eu,
    0xADu, 0xF7u, 0x87u, 0x7Cu, 0x9Cu, 0x57u, 0x3Bu, 0x32u,
    0x79u, 0x51u, 0x3Bu, 0x05u, 0x80u, 0x21u, 0xB8u, 0x21u,
    0xCAu, 0x49u, 0x53u, 0x93u, 0xA2u, 0x22u, 0x9Bu, 0x10u,
    0x68u, 0xA1u, 0xE5u, 0x8Eu, 0xE8u, 0x02u, 0xE9u, 0x29u,
    0x62u, 0xCFu, 0x05u, 0xB1u, 0xBBu, 0xB9u, 0xCFu, 0x07u,
    0xF5u, 0x75u, 0x4Du, 0x16u, 0x07u, 0xA0u, 0xCCu, 0xB7u,
},
.expData =
{
    0x01u, 0x00u, 0x01u, 0x00u,
},
.barrettData =
{
    0xDCu, 0x40u, 0x33u, 0x8Au, 0x4Eu, 0xA6u, 0xF1u, 0x08u,
    0x2Au, 0x7Bu, 0x5Cu, 0x77u, 0xC9u, 0xB1u, 0x95u, 0x68u,
    0x94u, 0xF2u, 0x80u, 0x0Eu, 0xA7u, 0x99u, 0xE5u, 0xBDu,
    0x07u, 0x91u, 0x0Cu, 0x66u, 0x62u, 0x3Du, 0x1Eu, 0x02u,
    0x6Cu, 0x12u, 0x3Bu, 0x79u, 0xE0u, 0xB6u, 0x81u, 0xB4u,
    0xACu, 0x85u, 0x75u, 0x44u, 0x95u, 0x0Du, 0xC7u, 0xE9u,
    0x69u, 0x7Du, 0xD3u, 0x30u, 0x4Bu, 0x57u, 0x4Du, 0x2Fu,
    0x6Eu, 0x80u, 0x51u, 0xC0u, 0x72u, 0x4Bu, 0x23u, 0x76u,
    0x82u, 0x91u, 0x98u, 0x47u, 0xFEu, 0x4Fu, 0xBCu, 0xBFu,
    0xA5u, 0x84u, 0x26u, 0xF0u, 0x90u, 0x62u, 0xC1u, 0x0Fu,
    0xFAu, 0x81u, 0xBAu, 0x57u, 0xDFu, 0x98u, 0x00u, 0xE3u,
}

```

5 PSoC™ 6 application notes

DRAFT

```
0xC6u, 0xACu, 0x99u, 0x82u, 0xFAu, 0x29u, 0x61u, 0xF3u,  
0x37u, 0x7Au, 0x61u, 0x09u, 0x25u, 0x92u, 0xCFu, 0xDFu,  
0x17u, 0x20u, 0x46u, 0x8Du, 0xBFu, 0x88u, 0xE7u, 0x0Bu,  
0xB5u, 0xAFu, 0xCEu, 0x03u, 0x8Au, 0xEAu, 0x33u, 0xC4u,  
0x8Cu, 0x1Bu, 0x44u, 0x41u, 0xC6u, 0x9Au, 0xCDu, 0x57u,  
0x5Fu, 0x59u, 0x6Eu, 0x1Eu, 0x1Cu, 0xDBu, 0xD7u, 0x37u,  
0x38u, 0x98u, 0xF6u, 0x0Bu, 0x3Du, 0xCDu, 0x11u, 0xA5u,  
0xF0u, 0x1Fu, 0x13u, 0x3Eu, 0x46u, 0x0Bu, 0xADu, 0x07u,  
0xA3u, 0x6Fu, 0x8Fu, 0xD5u, 0xCEu, 0xD8u, 0xA6u, 0x36u,  
0x8Eu, 0x39u, 0xDCu, 0xDCu, 0x07u, 0x6Fu, 0xE8u, 0x3Au,  
0x64u, 0x71u, 0x10u, 0xE1u, 0xCDu, 0x20u, 0xDFu, 0x4Bu,  
0xC8u, 0xA3u, 0x1Bu, 0x89u, 0x20u, 0x35u, 0x51u, 0x8Fu,  
0xA6u, 0x48u, 0x1Au, 0xF5u, 0xD5u, 0xF2u, 0x65u, 0x8Fu,  
0x3Au, 0x55u, 0x3Fu, 0x7Eu, 0x4Bu, 0x44u, 0x7Fu, 0xBAu,  
0x27u, 0xF4u, 0x19u, 0x2Cu, 0x53u, 0x06u, 0x75u, 0xB8u,  
0xB8u, 0xC4u, 0x8Fu, 0xCBu, 0xC9u, 0xE5u, 0xFBu, 0x91u,  
0xAAu, 0x4Du, 0x3Bu, 0xD2u, 0xA3u, 0x2Au, 0x36u, 0x2Fu,  
0xC9u, 0xBFu, 0xCCu, 0x8Du, 0xF9u, 0x3Eu, 0x5Fu, 0x9Eu,  
0xEEu, 0x19u, 0xF0u, 0xD5u, 0x58u, 0xE0u, 0x07u, 0x1Bu,  
0xBDu, 0xF1u, 0x42u, 0xF9u, 0xB2u, 0xC9u, 0x07u, 0x3Cu,  
0x44u, 0x67u, 0xD5u, 0x32u, 0x00u, 0x14u, 0x90u, 0x64u,  
0x01u, 0x00u, 0x00u, 0x00u,  
,  
.inverseModuloData =  
{  
    0x05u, 0xD9u, 0x5Au, 0x11u, 0xBDu, 0x82u, 0x6Au, 0x74u,  
    0x24u, 0x61u, 0xBBu, 0x30u, 0x03u, 0x6Du, 0x5Bu, 0xEDu,  
    0x61u, 0xCEu, 0x5Cu, 0x32u, 0xBBu, 0x1Du, 0x3Fu, 0x38u,  
    0xB8u, 0x75u, 0x36u, 0x1Au, 0x6Cu, 0x2Du, 0x46u, 0x3Cu,  
    0x1Au, 0x61u, 0xE1u, 0x63u, 0x2Cu, 0x8Fu, 0x49u, 0x80u,  
    0xCAu, 0xFFu, 0x51u, 0x5Fu, 0xC6u, 0x2Au, 0x2Au, 0x38u,  
    0xCFu, 0x6Du, 0x35u, 0x87u, 0xBCu, 0x74u, 0x47u, 0x2Fu,  
    0xE5u, 0x7Fu, 0xC3u, 0x18u, 0x8Du, 0x9Au, 0x60u, 0xCAu,  
    0xEFu, 0x84u, 0x2Eu, 0xF2u, 0x6Eu, 0x8Du, 0x88u, 0xC3u,  
    0x13u, 0x9Du, 0x4Eu, 0x81u, 0x34u, 0xFDu, 0x21u, 0x18u,  
    0xDDu, 0xE7u, 0xD3u, 0x71u, 0x51u, 0x49u, 0x6Eu, 0xF9u,  
    0x24u, 0xAFu, 0x4Eu, 0x94u, 0x23u, 0xD8u, 0x05u, 0x64u,  
    0x42u, 0x74u, 0x48u, 0x02u, 0x54u, 0x8Bu, 0xE4u, 0x7Bu,  
    0xA9u, 0x69u, 0x3Du, 0x56u, 0xF0u, 0xD1u, 0xA0u, 0x39u,  
    0x86u, 0x11u, 0x1Eu, 0xEFu, 0x0Bu, 0x64u, 0x60u, 0x7Du,  
    0x32u, 0x8Fu, 0x4Bu, 0x01u, 0xC6u, 0x8Eu, 0x84u, 0x8Du,  
    0xAFu, 0xD6u, 0x1Du, 0x0Fu, 0x1Cu, 0x38u, 0x15u, 0x4Bu,  
    0xF3u, 0x8Cu, 0xB1u, 0xF0u, 0x05u, 0x96u, 0xFAu, 0x97u,  
    0x22u, 0x4Eu, 0x2Du, 0x6Bu, 0xDBu, 0x7Du, 0x31u, 0x92u,  
    0x8Eu, 0x3Bu, 0x33u, 0x5Bu, 0xA2u, 0xFDu, 0xF8u, 0x12u,  
    0x55u, 0x15u, 0xD3u, 0x39u, 0xE3u, 0x83u, 0x48u, 0xA8u,  
    0x02u, 0x46u, 0x40u, 0x17u, 0x92u, 0x4Du, 0x89u, 0x6Du,  
    0x86u, 0xCCu, 0x40u, 0x07u, 0x16u, 0x82u, 0x68u, 0xE7u,  
    0x28u, 0xB6u, 0xF9u, 0x52u, 0xFCu, 0x8Fu, 0x84u, 0x84u,  
    0x4Bu, 0xBBu, 0x7Bu, 0x20u, 0xEEu, 0x3Du, 0x14u, 0x26u,  
    0x5Eu, 0x17u, 0xC7u, 0xFFu, 0x4Eu, 0xBAu, 0xAEu, 0x81u,  
    0x36u, 0x1Du, 0x10u, 0xE1u, 0x37u, 0x25u, 0xBEu, 0x02u,  
    0x84u, 0x09u, 0x54u, 0xC8u, 0xD5u, 0x09u, 0x78u, 0xD4u,
```

5 PSoC™ 6 application notes

DRAFT

```
0x34u, 0x4Au, 0x03u, 0x5Bu, 0xA3u, 0x6Du, 0xEEu, 0x36u,
0xACu, 0xF2u, 0xE9u, 0x3Eu, 0x67u, 0xF4u, 0xD0u, 0xA5u,
0x1Cu, 0x32u, 0xDEu, 0x56u, 0x84u, 0x5Cu, 0xB6u, 0xA7u,
0xE3u, 0x3Du, 0xF6u, 0xC2u, 0x44u, 0xFDu, 0xFAu, 0xC0u,
},
.rBarData =
{
    0xCDu, 0x0Bu, 0xC9u, 0xF6u, 0xFFu, 0x9Eu, 0xE8u, 0x5Eu,
    0x19u, 0xBDu, 0xBDu, 0x68u, 0x89u, 0xD5u, 0x54u, 0x10u,
    0x2Cu, 0xD8u, 0x5Fu, 0x75u, 0xA4u, 0xE5u, 0xD4u, 0xD1u,
    0x75u, 0xEFu, 0x17u, 0x33u, 0x1Bu, 0xA8u, 0x88u, 0xABu,
    0x9Fu, 0x77u, 0x4Fu, 0xFEu, 0x02u, 0x86u, 0x81u, 0x15u,
    0xCFu, 0x90u, 0x71u, 0xD3u, 0x86u, 0xE3u, 0x34u, 0x24u,
    0x38u, 0xFDu, 0x5Fu, 0xB3u, 0xE1u, 0x8Au, 0xFFu, 0x33u,
    0xB6u, 0xE1u, 0x44u, 0x91u, 0xE1u, 0x95u, 0x05u, 0x9Bu,
    0x25u, 0x5Fu, 0x4Eu, 0xBEu, 0x8Fu, 0x02u, 0x5Fu, 0xB8u,
    0x7Du, 0xD8u, 0x65u, 0x0Du, 0x0Cu, 0xCBu, 0x48u, 0x14u,
    0x6Du, 0xCCu, 0xA1u, 0xD3u, 0xC8u, 0x23u, 0xF7u, 0x73u,
    0x8Fu, 0x4Fu, 0xCEu, 0xE4u, 0x0Eu, 0x1Bu, 0xAEu, 0xFCu,
    0xB1u, 0x39u, 0x06u, 0x40u, 0x41u, 0xCCu, 0x08u, 0x47u,
    0x60u, 0xE5u, 0x7Cu, 0x26u, 0x90u, 0x1Fu, 0x1Fu, 0xF3u,
    0x3Au, 0xC7u, 0xFFu, 0x1Bu, 0x48u, 0xA4u, 0x89u, 0xA8u,
    0x9Bu, 0x7Fu, 0xE0u, 0xCEu, 0x19u, 0x8Cu, 0x78u, 0x7Fu,
    0x38u, 0x73u, 0x4Fu, 0xBBu, 0x38u, 0xC6u, 0xE6u, 0x39u,
    0x71u, 0xCDu, 0xF5u, 0x7Du, 0xD5u, 0x46u, 0xB3u, 0xF3u,
    0x3Au, 0xC2u, 0x9Bu, 0x36u, 0x6Eu, 0x31u, 0x9Du, 0x7Du,
    0xDEu, 0x03u, 0x05u, 0xD3u, 0xE1u, 0x4Eu, 0x2Fu, 0xABu,
    0xD8u, 0xBAu, 0xD1u, 0xB9u, 0xB1u, 0x10u, 0xB5u, 0xADu,
    0x3Eu, 0x70u, 0x44u, 0x50u, 0x42u, 0x01u, 0x41u, 0x20u,
    0xEAu, 0xD9u, 0x24u, 0xC8u, 0xDBu, 0x7Au, 0xEAu, 0x73u,
    0x02u, 0x26u, 0x14u, 0x80u, 0xE8u, 0xCFu, 0xC4u, 0x6Au,
    0xEAu, 0xA6u, 0x11u, 0x13u, 0x5Eu, 0x14u, 0xB0u, 0x35u,
    0x2Eu, 0x9Fu, 0xDEu, 0x1Fu, 0xE4u, 0x29u, 0x46u, 0xF1u,
    0x52u, 0x08u, 0x78u, 0x83u, 0x63u, 0xA8u, 0xC4u, 0xCDu,
    0x86u, 0xAEu, 0xC4u, 0xFAu, 0x7Fu, 0xDEu, 0x47u, 0xDEu,
    0x35u, 0xB6u, 0xACu, 0x6Cu, 0x5Du, 0xDDu, 0x64u, 0xEFu,
    0x97u, 0x5Eu, 0x1Au, 0x71u, 0x17u, 0xFDu, 0x16u, 0xD6u,
    0x9Du, 0x30u, 0xFAu, 0x4Eu, 0x44u, 0x46u, 0x30u, 0xF8u,
    0x0Au, 0x8Au, 0xB2u, 0xE9u, 0xF8u, 0x5Fu, 0x33u, 0x48u,
},
};

#if defined(__cplusplus)
}
#endif
```

~~DO NOT USE~~ 5 PSoC™ 6 application notes

5.15.9 Appendix C - Debug port access settings

Table 130 shows the memory location for each of the three type of debug access restrictions (SECURE, DEAD, and NORMAL). SARs and DARs are stored in eFuse, but NARs are stored in SFlash. For eFuse, this is only the read location; the eFuse write location is in a different memory location and set up as one byte per bit.

Table 130 Location of access restrictions

Access restriction	ACCESS_RESTRICT0 (ADDR)	ACCESS_RESTRICT1 (ADDR)	Storage area
SECURE	0x402C_0829*	0x402C_082A*	eFuse
DEAD	0x402C_0827*	0x402C_0828*	eFuse
NORMAL	0x1600_1A00	0x1600_1A01	SFlash

Note: *For eFuse, this is the read address only. When writing to eFuse, each bit is programmed with a byte location which is different from the byte read location.*

5.15.9.1 Debug access settings

The format for all three types of access restrictions (SECURE, DEAD, NORMAL) is the same, although stored in different locations. The default state of all debug access restrictions is zero, which means that all debug ports are open for full access.

This section defines the location and definition of each of those parameters. [Figure 560](#) and [Figure 561](#) represent the format for the 2-byte access restriction structure.

	Bit	7	6	5	4	3	2	1	0	
Field	MMIO_Allowed [7:6] (SYS_AP)	SFlash_Allowed [5:4] (SYS_AP)	SYS_AP_MPUI Enable	SYS_AP Disable	CM4 Disable	CM0 Disable				

Figure 560 ACCESS_RESTRICT0

	Bit	7	6	5	4	3	2	1	0	
Field	DIRECT_EXECUTE (SYS_AP)	SMIF_XIP (SYS_AP)	SRAM_ALLOWED[5:3] (SYS_AP)	FLASH_ALLOWED[2:0] (SYS_AP)						

Figure 561 ACCESS_RESTRICT1

[Table 131](#) Access restriction parameters.

Field	Value	Description
MMIO_Allowed	0x0: All MMIO register 0x1: Only IPC ports 0, 1, and 2 0x2 or 0x3: No MMIO access	Defines what MMIO register are accessible via the SYS_AP. IPC ports 0, 1, and 3 are used for system calls required for programming of the device.

5 PSoC™ 6 application notes

Field	Value	Description
SFlash_Allowed	0: Entire SFlash accessible 1: Bottom ½ of SFlash accessible 2: Bottom ¼ of SFlash accessible 3: No access allowed to SFlash	This field indicates what portion of the SFlash main region is accessible through the SYS_AP. Only a portion of flash starting at the bottom of the area is exposed. Valid only if SYS_DISABLE=0 and SYS_AP_MP_ENABLE=1.
SYS_AP_MP_ENABLE	0x0: SYS_AP MPU disabled 0x1: SYS_AP MPU enabled	SYS_AP_DISABLE must not be disabled for the MPU to be enabled.
SYS_AP_DISABLE	0x0: SYS_AP not disabled 0x1: SYS_AP disabled	Disables the SYS_AP
CM4_DISABLE	0x0: CM4_AP not disabled 0x1: CM4_AP disabled	Disables the CM4_AP
CM0_DISABLE	0x0: CM0_AP not disabled 0x1: CM0_AP disabled	Disables the CM0_AP
DIRECT_EXE_DISABLE	0x0: Not disabled 0x1: Disable	Disable Direct Execute system call functionality.
SMIF_XIP_ALLOWED	0x0: Entire Region 0x1: Nothing	This field indicates what portion of XIP is accessible through the system access port.
SRAM_ALLOWED	0x0: entire region 0x1: 7/8 0x2: 3/4th 0x3: 1/2 0x4: 1/4th 0x5: 1/8th 0x6: 1/16th 0x7: nothing	This field indicates what portion of the SRAM region is accessible through the SYS_AP. Only a portion of SRAM starting at the bottom of the area is exposed. Valid only if SYS_DISABLE=0 and SYS_AP_MP_ENABLE=1.
FLASH_ALLOWED	0x0: entire region 0x1: 7/8 0x2: 3/4th 0x3: 1/2 0x4: 1/4th 0x5: 1/8th 0x6: 1/16th 0x7: nothing	This field indicates what portion of the flash main region is accessible through the SYS_AP. Only a portion of flash starting at the bottom of the area is exposed. Valid only if SYS_DISABLE=0 and SYS_AP_MP_ENABLE=1.

5.15.9.2 Firmware control of Debug Port

If any of the three debug access ports (CM0+, CM4, SYS) are disabled in the SECURE life cycle stage, there is no way to connect to those ports. If any of the debug ports are not disabled in the SECURE life cycle stage, it is possible for the debug port to be opened either automatically or with firmware. You can disable any combination of the three access ports, and the ports not disabled may be connected to an external

~~5 PSoC™ 6 application notes~~

programmer/debugger. The 2nd generation parts are slightly different than the 1st generation parts as described in the following sections.

~~5.15.9.2.1 1st generation PSoC™ 6 devices~~

In the SECURE life cycle stage, if the debug access ports were not disabled, a debugger/programmer cannot connect to the enabled access port until the GPIOs used for debugging are configured properly. This can be accomplished within the user's firmware with the code shown in [section 9.2.3](#).

5.15.9.2.2 2nd generation PSoC™ 6 devices

The 2nd generation parts have an additional option for configuring the debug ports. A flag in TOC2 Flash boot options, allow the user to set the default state of the debug ports when in the SECURE life cycle stage. If the “SWJ pin state” bits are set to 0x10, the GPIOs will be automatically configured to allow connection by a debugger/programmer hardware without additional user firmware. See the table below for the TOC2 Flash boot flags for the SWJ pin state.

Table 131 Excerpt from TOC2 Flash boot flags

SWJ (debug) pin state	[6:5]	0x00 = Do not enable SWJ pins 0x01 = Do not enable SWJ pins 0x10 = Enable SWJ pins 0x11 = Do not enable SWJ pins	Determines whether SWJ pins are configured in SWJ mode by Flash boot. <i>Note:</i> <i>SWJ pins may be enabled later in the user code.</i>
-----------------------	-------	--	--

~~5 PSoC™ 6 application notes~~

~~5.15.9.2.3 Configure SWJ for Debug~~

The code snippet below is used to configure the GPIO pins used for debugging and programming of the device. It will work for all PSoC™ 62/63 devices and allow the debugger/programmer to connect to any debug access ports not disabled.

```

static void configure_swj(void)
{
    /* Enable CM0+, CM4 and System Access Ports */
    CPUSS_AP_CTL = (CY_FB_AP_CTL_CM0_ENABLE_MASK | CY_FB_AP_CTL_CMx_ENABLE_MASK |
CY_FB_AP_CTL_SYS_ENABLE_MASK);

    /* Note, PSoC6A-BLE-2 and PSoC6A-2M devices use the same pins and pin configuration
     * for SWJ functionality
     * P6_4 - SWO/TDO
     * P6_5 - SWDOE/TDI
     * P6_6 - SWDIO/TMS
     * P6_7 - SWCLK/TCLK
     */
    Cy_GPIO_Pin_FastInit(P6_4_PORT, P6_4_NUM, CY_GPIO_DM_STRONG_IN_OFF, 0,
P6_4_CPUSS_SWJ_SWO_TDO);
    Cy_GPIO_Pin_FastInit(P6_5_PORT, P6_5_NUM, CY_GPIO_DM_PULLUP_IN_OFF, 0,
P6_5_CPUSS_SWJ_SWDOE_TDI);
    Cy_GPIO_Pin_FastInit(P6_6_PORT, P6_6_NUM, CY_GPIO_DM_PULLUP_IN_OFF, 0,
P6_6_CPUSS_SWJ_SWDIO_TMS);
    Cy_GPIO_Pin_FastInit(P6_7_PORT, P6_7_NUM, CY_GPIO_DM_PULLDOWN_IN_OFF, 0,
P6_7_CPUSS_SWJ_SWCLK_TCLK);
}

```

5.15.9.3 eFuse programming for debug access restrictions and lifecycle stage

A byte of data is required to program each bit of the eFuse. The following pattern is used to program, validate, or set as ‘don’t care’ each bit of eFuse.

```

/* EFUSE bit action macros */
#define CY_EFUSE_STATE_SET      (0x01U) /* Tell programmer to set the EFUSE bit */
#define CY_EFUSE_STATE_UNSET    (0x00U) /* Tell programmer to check that the EFUSE bit is not
set */
#define CY_EFUSE_STATE_IGNORE   (0xffU) /* Tell programmer to ignore the EFUSE bit */

```

The following code snippet is an example of how to set the SARs by programming the eFuse for the 1st generation parts. The full source is available in the *mtb-psoc6-example-security/proj_btldr_cm0p/source/*

5 PSoC™ 6 application notes

~~DRAFT~~
cy_ps_efuse.c file. Note that this code snippet also sets the device to move to the SECURE lifecycle stage, by setting the SECURE bit toward the bottom of the structure.

```

/* EFUSE configuration */
CY_SECTION(".cy_efuse") __USED const cy_stc_efuse_data_t cy_efuseData =
{
    .RESERVED = CY_EFUSE_RESERVED0,                                /* Reserved bits ignored */

    /* Dead access restrictions are set in this section */
    /* CM40+ and CM4 debug are disabled, but SYS_AP is left open. */
    .DEAD_ACCESS_RESTRICT0 =
    {
        .CM0_DISABLE      = CY_EFUSE_STATE_SET,                  /* Disable CM0+ access port */
        .CM4_DISABLE      = CY_EFUSE_STATE_SET,                  /* Disable CM4 access port */
        .SYS_DISABLE      = CY_EFUSE_STATE_SET,                  /* Enable System access port */
        .SYS_AP_MPU_ENABLE = CY_EFUSE_STATE_UNSET,               /* Enable the system access port
MPU */
        .SFLASH_ALLOWED   = CY_EFUSE_SFLASH_ALLOWED_ENTIRE,     /* SYS AP MPU protection of
SFlash */
        .MMIO_ALLOWED     = CY_EFUSE_MMIO_ALLOWED_ENTIRE,       /* SYS AP MPU protection of MMIO
*/
    },
    .DEAD_ACCESS_RESTRICT1 =
    {
        .FLASH_ALLOWED    = CY_EFUSE_FLASH_ALLOWED_ENTIRE,     /* SYS AP MPU protection of Flash
*/
        .SRAM_ALLOWED     = CY_EFUSE_SRAM_ALLOWED_ENTIRE,       /* SYS AP MPU protection of SRAM
*/
        .SMIF_XIP_ALLOWED = CY_EFUSE_SMIF_XIP_ALLOWED_ENTIRE,  /* SYS AP MPU protection of SMIF
XIP */
        .DIRECT_EXECUTE_DISABLE = CY_EFUSE_STATE_UNSET         /* Disable "direct execute"
system call */
    },
    /* Secure access restrictions are set in this section */
    /* All debug ports are disabled here. */

    .SECURE_ACCESS_RESTRICT0 =
    {
        .CM0_DISABLE      = CY_EFUSE_STATE_SET,                  /* Disable CM0+ access port */
        .CM4_DISABLE      = CY_EFUSE_STATE_SET,                  /* Disable CM4 access port */
        .SYS_DISABLE      = CY_EFUSE_STATE_SET,                  /* Enable System access port */
        .SYS_AP_MPU_ENABLE = CY_EFUSE_STATE_UNSET,               /* Enable the system access port
MPU */
        .SFLASH_ALLOWED   = CY_EFUSE_SFLASH_ALLOWED_ENTIRE,     /* SYS AP MPU protection of
SFlash */
        .MMIO_ALLOWED     = CY_EFUSE_MMIO_ALLOWED_ENTIRE,       /* SYS AP MPU protection of MMIO
*/
    },
    .SECURE_ACCESS_RESTRICT1 =
    {
        .FLASH_ALLOWED    = CY_EFUSE_FLASH_ALLOWED_ENTIRE,     /* SYS AP MPU protection of Flash
*/
    }
};

```

5 PSoC™ 6 application notes

```

/* DRAFT
    .SRAM_ALLOWED      = CY_EFUSE_SRAM_ALLOWED_ENTIRE,      /* SYS AP MPU protection of SRAM
*/
    .SMIF_XIP_ALLOWED = CY_EFUSE_SMIF_XIP_ALLOWED_ENTIRE, /* SYS AP MPU protection of SMIF
XIP */
    .DIRECT_EXECUTE_DISABLE = CY_EFUSE_STATE_UNSET        /* Disable "direct execute"
system call */
},
/* This section sets the Life cycle to SECURE */
/* You can only set either the "SECURE_WITH_DEBUG" or the "SECURE" bit but not both */
.LIFE_CYCLE_STAGE =
{
    .NORMAL          = CY_EFUSE_STATE_IGNORE, /* Normal life cycle already set - ignore */
    .SECURE_WITH_DEBUG = CY_EFUSE_STATE_IGNORE, /* Secure with Debug life cycle - Ignore */
    .SECURE          = CY_EFUSE_STATE_SET,     /* Transition to SECURE */
    .RMA             = CY_EFUSE_STATE_IGNORE, /* Transition to RMA - Ignore */
    .RESERVED        = CY_EFUSE_LIFE_CYCLE_RESERVED0 /* Reserved bits ignored */
},
.RESERVED1 = CY_EFUSE_RESERVED1,           /* Reserved bits ignored */
.CUSTOMER_DATA =
{
    CY_EFUSE_CUSTOMER_IGNORE512            /* All user EFUSE data ignored */
}
};

```

Table 132 eFuse parameters

Parameter	Description
CM0_DISABLE	Disables debug access to CM0+
CM4_DISABLE	Disables debug access to CM4
SYS_DISABLE	Disables debug access to the system access port
SYS_AP_MPU_ENABLE	Enables MPU on the system access port
SFLASH_ALLOWED	Enables access to the SFlash; enabled by default
MMIO_ALLOWED	Enables access to the MMIO registers
FLASH_ALLOWED	Enables access to the flash
SRAM_ALLOWED	Enables access to the SRAM
SMIF_XIP_ALLOWED	Enables execution from the external SMIF memory
DIRECT_EXECUTE_DISABLE	Disables the "direct execute" system call

When you are ready to advance to the SECURE lifecycle stage, the device must be in the NORMAL lifecycle stage. This is because it is not possible to move from the SECURE_WITH_DEBUG lifecycle stage to the SECURE lifecycle stage.

1. Change the line “.SECURE = CY_EFUSE_STATE_IGNORE” to “.SECURE = CY_EFUSE_STATE_SET” to set the SECURE bit.
2. Set CY_EFUSE_AVAILABLE to 1 in *mtb-example-psoc6-security/proj_btldr_cm0p/source/cy_ps_efuse.h*
 - #define CY_EFUSE_AVAILABLE 1.
3. Rebuild the project and program the part. Ensure that the device VDDIO0 pin must be at 2.5 V.

5 PSoC™ 6 application notes

Important notes:

Infineon provides the CYPRESS™ Programmer application to work with the MiniProg4 Programmer/Debugger dongle to program and/or debug the PSoC™ 6 devices. 1st and 2nd generation parts have different processes to move a device to the SECURE stage. Cypress Programmer makes these differences transparent to the user.

During the programming operation of PSoC™ 6 devices, CYPRESS™ Programmer detect when the SECURE eFuse bit is going to be set.

For both the 1st and 2nd generation parts, steps a, b and c below must occur. For the 1st generation parts, CYPRESS™ Programmer must perform these steps. The 2nd generation parts have an extra system call “Transition2Secure” that performs these three steps automatically inside the device.

1. Validate the parts of the SFlash area (trim values and Flash boot) with the internal Factory_HASH to ensure that the part has not been modified after leaving the factory.
2. Generate Secure_HASH based on the TOC2 entries. By default, Secure_HASH includes all of SFlash. Additional areas from the user flash may be included if additional entries are added in TOC2. This hash is written into the Secure_HASH area of the eFuse along with the number of zeros in the hash. This guarantees that the hash cannot be modified by simply changing zeros to ones.
3. Program the eFuse bits for access restrictions and the SECURE bit. This is done with eFuse programming system calls; you cannot write directly to eFuse bits.

See the [PSoC™ 6 Programming Specifications](#) for more information.

~~5 PSoC™ 6 application notes~~

~~DRAFT~~ 5.15.10 Appendix D - Transition to RMA

To transition a device to the RMA stage, you must have access to the following:

- The device unique ID
- Private key that is paired with a public key stored and authenticated in SFlash.

Send the following special commands from outside the device via UART, SPI, I²C, etc.:

1. Read the internal unique device ID.
2. Invoke the transition to RMA.

You can implement the special commands in two ways:

1. Include these special commands as part of the existing application.
2. Create a special device code image that supports only the required commands.
 - This is a safer and secured way send the special commands. With this approach, when bootloading this special code image, all proprietary and sensitive data can be erased at the same time. In addition, a special code image allows an easy implementation of a standard interface such as a UART to implement the communication needed to invoke the commands.

After this infrastructure is in place, do the following to transition the device to RMA mode.

1. Erase all sensitive or proprietary code stored in the device. This may be performed with a special command or with a special code image described above. Erase the flash at least four times to ensure there is no way to detect any residual code. The public key stored in SFlash must remain because it is used to transition to the RMA lifecycle stage and to allow Infineon to open the RMA later.
2. Read the device unique ID stored in the device's SFlash. This can be done by invoking the 0x1F (Read Unique ID) system call, and then sending ID out via the communication interface.
3. Generate a certificate using the unique ID and the customer's private key that is paired with the public key stored internally in SFlash. This is the same method that is used to sign code as described in Section 3.5, [Code signing and verification](#), using the same private/public key pairs. The format of the certificate is shown in [Figure 562](#).

Object size in bytes of the certificate 0x00000114 (4 bytes)
Command ID 0x28000000 (4 bytes)
Unique ID (10 bytes)
Zero Padding 0x0000 (2 bytes)
Digital Signature (256 bytes)

Figure 562 RMA certificate format

4. Send a command to the device that includes the certificate generated. You must implement code to accept this certificate to invoke the transition to RMA system call (0x28) and pass the certificate as its parameter. The device V_{DDIO0} supply must be at 2.5V before performing this step, because the RMA eFuse is to be programmed. (Any programming of eFuse bits requires the V_{DDIO0} to be 2.5 V.)
5. After the device is reset or power cycled, it will sit idle awaiting a single command from the debug port to open RMA (system call 0x29) along with the same certificate that was used to invoke the transition to RMA in the first place. It will have all the same access modes as Virgin mode, but a debugger/programmer must invoke the open RMA system call every time the device is reset or power cycled. The device in this state is unusable except for failure analysis.

After you have performed these steps, you can send the device and certificate to Infineon to allow failure analysis. Note that this special certificate is valid only for the one part for which it was generated.

5 PSoC™ 6 application notes

5.15.11 Appendix E - Protection unit configuration

5.15.11.1 Example Protection unit configuration

This is an example how the SMPUs are configured in the example project for the 1 M flash device.

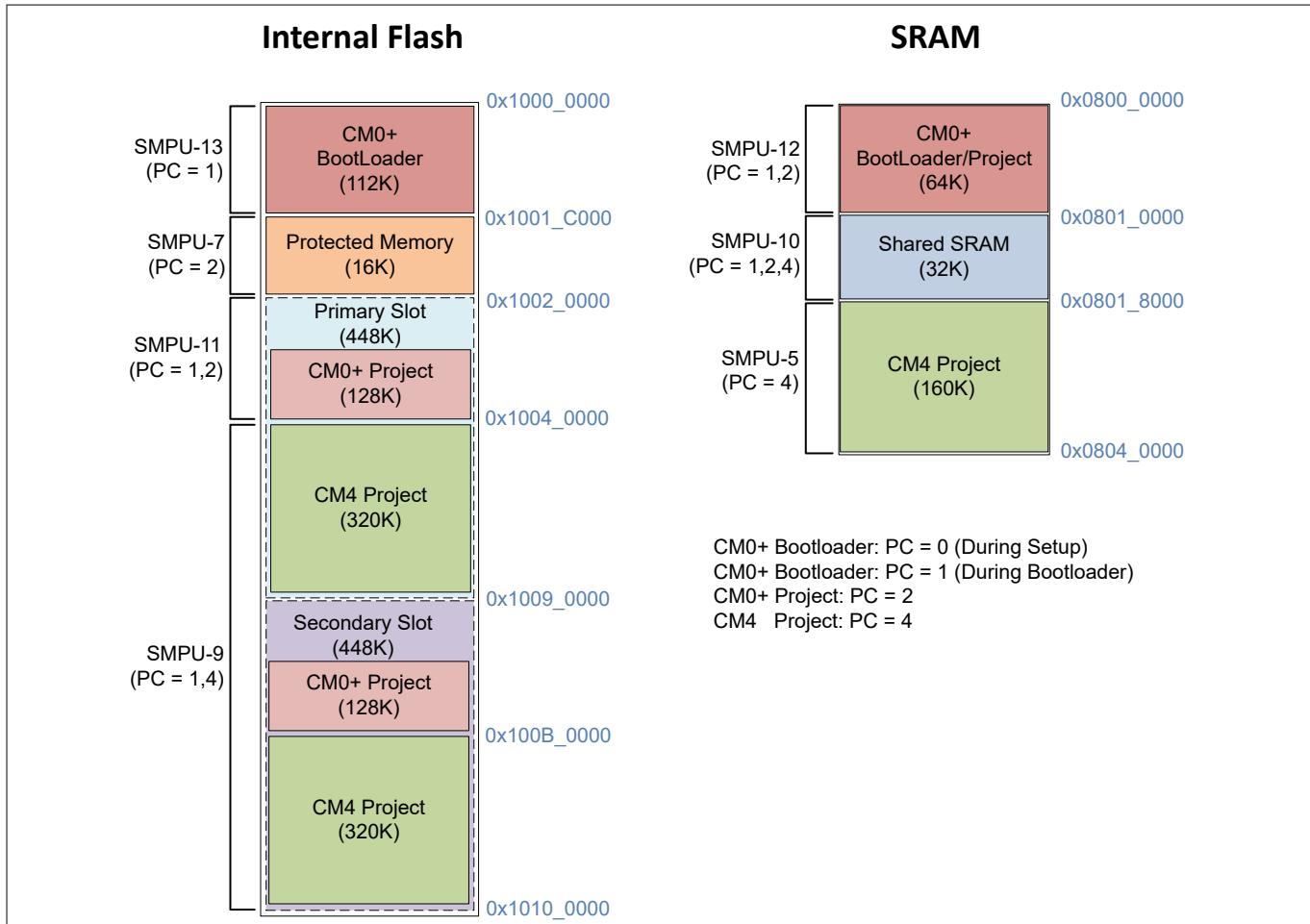


Figure 563 Protection unit configuration

Table 133 Summary of SMPU settings

Section	Bus master	Memory	Protection context	Start address	Size	Access attributes	SMPU
Bootloader	CM0+	Flash	PC = 1,2 ²	0x1000_000 0	0x0001_C00 0 (112K)	R/X	13
CM0+ Project	CM0+	Flash	PC = 1,2	0x1002_000 0	0x0002_000 0 (128K)	R/W/X	11
CM4 Project	CM4	Flash	PC = 1,4	0x1004_000 0	0x000C_000 0 (768K)	R/W/X	9
Protected Flash	CM0+	Flash	PC = 2	0x1001_C00 0	0x0000_400 0 (16K)	R/W/X	7

(table continues...)

5 PSoC™ 6 application notes

DRAFT
Table 133 (continued) Summary of SMPU settings

Section	Bus master	Memory	Protection context	Start address	Size	Access attributes	SMPU
CM0+ Project/ Bootloader SRAM	CM0+	SRAM	PC = 1,2	0x0800_000 0	0x0001_000 0 (64K)	R/W/X ³	12 ¹
Shared SRAM	CM0+/CM4	SRAM	PC = 1, 2, 4	0x0801_000 0	0x0000_800 0 (32K)	R/W	10
CM4 Project SRAM	CM4	SRAM	PC = 4	0x0804_000 0	0x0002_800 0 (160K)	R/W/X ⁴	5

Notes:

1. The bootloader and the CM0+ runtime project uses the same SRAM and SMPU because they will never run at the same time. The runtime project actually can reuse the bootloader SRAM.
2. The bootloader needs to be PC=1,2 since it has to transition from PC=1 to PC=2 when the boot loader jumps to the CM0+ project.
3. The CM0+ SRAM must be executable since when programming Flash, it must run in SRAM in case the code execution is in the same flash block as the one being updated.
4. The CM4 SRAM must be executable since when programming Flash, it must run in SRAM in case the code execution is in the same flash block as the one being updated.

To maximize security, ensure the following order for programming the protection units. By default, both CM0+ and CM4 are set to PC=0.

1. Make all protection unit configuration changes while CM0+ is still in PC=0 mode.
2. After configuring the protection units, change CM0+ PC value to a non-zero value.
3. Set the protection context (PC) mask values for CM0+ and CM4. The mask values determine the PC values that each of these bus masters can switch to. Note that the mask value does not set the current PC value.
4. Set the CM4 protection context to a non-zero value. In this example, it is set to PC=4.
5. Configure the SMPU slave protection unit structures and enable them. Create a structure for each SMPU. Use the `Cy_Prot_ConfigSmpuSlaveStruct()` function to configure the SMPU, and the `Cy_Prot_EnableSmpuSlaveStruct()` function to enable it.
6. Configure the master protection unit structures with the `Cy_Prot_ConfigSmpuMasterStruct()` function for all SMPUs. Enable the master structs with the `Cy_Prot_EnableSmpuMasterStruct()` function.
7. Program the protection units slave structures to be owned by PC=0.
8. Protect the MS_CTL registers so that the bus master PC masks cannot be altered.
9. Use PPUs to secure the bus master registers (both master and slave), and enable them.
10. Change the protection context for CM0+ to a non-zero value; in this example PC=1.

Note: See the `proj_btldr_cmp0/cy_ps_prot_units.c` file in CE234992 for an example of code that configures the protection units.

5 PSoC™ 6 application notes

~~DRAFT~~ 5.15.11.2 Pre-configured protection units

Some protection units are configured during the boot process and must not be reconfigured. These protection units are vital to providing a secure system and providing a reliable access to system call functions.

It is important to note that if the protection context of any bus master, especially CM0+ and CM4, is left at 0 (PC=0), that bus master will have full access to all memory and registers no matter how the protection units are configured.

To achieve the best security, do not run any bus master at PC=0 after the configuration process is complete. This should be done before CM4 is enabled by CM0+ and right after all protection-associated registers are configured.

[Table 134](#) lists the protection units that must not be reconfigured by the user. These settings make sure that any modifications to eFuse or flash must go through system calls to provide proper security.

Table 134 Protection units used by the system

Protection unit	Usage description
SMPU 15	Read/write restriction for ROM private stack
SMPU 14	Read/write restriction for ROM region
PROG PPU 15	Write restriction for CPUSS_AP_CTL, PROTECTION, CM0_NMI_CTL, DP_CTL and MBIST_CTL registers
PROG PPU 14	Read/write restriction for CPUSS_WOUNDING and CM0_PC0_HANDLER registers
PROG PPU 13	Write restriction for FlashC_FM_CTL.BOOKMARK register
PROG PPU 12	Read/write restriction for eFuse region (excluding CUSTOMER_DATA)
PROG PPU 11	Write restriction for IPC 0, 1 and 2 during system calls
PROG PPU 10	Read/write restriction for Crypto during system calls that use crypto operations
PROG PPU 9	Read/write restriction for FM_CTL registers

5 PSoC™ 6 application notes

5.15.12 Appendix F - Debug codes for failed boot sequences

If the user application flash or the TOC2 was determined to be invalid, an error code will be written to IPC.DATA (structure 2). This allows you to detect the cause of failure during debug. [Table 135](#) lists all possible values.

Table 135 Error codes and values during debug

Error name	Value	Description
CY_FB_STATUS_SUCCESS	0xA100_0100	Success status value
CY_FB_STATUS_BUSY_WAIT_LOOP	0xA100_0101	Debugger probe acquired the device in Test mode. Flash boot has entered a busy wait loop.
CY_FB_ERROR_INVALID_APP_SIGN	0xF100_0100	Application signature validation failed for the device families where Flash boot launches only one application from TOC2. Either application structure or a digital signature is invalid for the device families for which Flash boot may launch either of two application in TOC2.
CY_FB_ERROR_INVALID_TOC	0xF100_0101	Empty or invalid TOC
CY_FB_ERROR_INVALID_KEY	0xF100_0102	Invalid public key
CY_FB_ERROR_UNREACHABLE	0xF100_0103	Unreachable code
CY_FB_ERROR_TOC_DATA_CLOCK	0xF100_0104	TOC contains an invalid CM0+ clock attribute
CY_FB_ERROR_TOC_DATA_DELAY	0xF100_0105	TOC contains an invalid listen window delay
CY_FB_ERROR_FLL_CONFIG	0xF100_0106	FLL configuration failed
CY_FB_ERROR_INVALID_APP0_DATA	0xF100_0107	Application structure is invalid for the device families where Flash boot may launch only one app from TOC2.
CY_FB_ERROR_CRYPTO	0xF100_0108	Error in crypto operation
CY_FB_ERROR_INVALID_PARAM	0xF100_0109	Invalid parameter value
CY_FB_ERROR_BOOT_HARD_FAULT	0xF100_010a	A hard fault exception occurred in Flash boot
CY_FB_ERROR_UNEXPECTED_INTERRUPT	0xF100_010B	An unexpected interrupt occurred in Flash boot
CY_FB_ERROR_BOOTLOADER	0xF100_0140	A bootloader error occurred
CY_FB_ERROR_BOOT_LIN_INIT	0xF100_0141	Bootloader error: LIN initialization failed
CY_FB_ERROR_BOOT_LIN_SET_CMD	0xF100_0142	Bootloader error: LinSetCmd() failed

5 PSoC™ 6 application notes**Related documents**

- - 1
 - 5
- DRAFT*

Table 136 Reference documents**Code examples**

[CE234992 PSoC™ 6 MCU: Security Application Template](#)

Application notes

AN221774	Getting started with PSoC™ 6 MCU
AN210781	Getting started with PSoC™ 6 MCU with Bluetooth® Low Energy connectivity
AN218241	PSoC™ 6 MCU hardware design considerations
AN213924	PSoC™ 6 MCU bootloader software development kit (SDK) guide

Device and support documentation

[PSoC™ 6 MCU datasheets](#)

[PSoC™ 6 MCU technical reference manuals](#)

[PSoC™ 6 MCU programming specification](#)

[CyMCUElfTool user guide](#)

Development kit (DVK) documentation

[CY8CKIT-062-BLE, PSoC™ 6 Bluetooth® LE pioneer kit](#)

[CY8CKIT-062-WIFI-BT, PSoC™ 6 Wi-Fi-Bluetooth® pioneer kit](#)

[CY8CPROTO-062-4343W, PSoC™ 6 Wi-Fi Bluetooth® prototyping kit](#)

[CY8CPROTO-063-BLE, PSoC™ 6 Bluetooth® LE prototyping kit](#)

~~5 PSoC™ 6 application notes~~

~~DRAFT~~ 5.15.13 Revision history

Document version	Date of release	Description of changes
**	2018-05-03	Initial release.
*A	2019-04-30	<p>Section 1– Specified the dual CPU 62/63 devices</p> <p>Section 2.2 – Expanded the definition of Protection Context (PC) in.</p> <p>Section 2.4.4 – Rephrased third sentence in for clarity</p> <p>Section 3 – Made minor changes in for clarity</p> <p>Figure 2– Fixed a minor typo in</p> <p>Table 1– Updated the description of “cy_si_config.c” to mention that system calls and crypto functions were included.</p> <p>Section 6.1.2 – Clarified that memory read and writes mentioned were through system calls.</p> <p>Section 6.3 – Minor re-wording of first paragraph, changed V_{DDD} to V_{DDIO0}</p> <p>Table 6 - Updated definition for SYS_AP_MPUEnable</p> <p>Other minor wording changes throughout document.</p> <p>Related documents– Added kits with links, added reference to AN221774, restructured device documentation</p>
*B	2021-03-16	Updated to Infineon template.
*C	2022-06-09	This is a total rewrite of this application note. All code examples are based on the code example CE234992 PSoC™ 6 MCU: Security Application Template .
*D	2022-08-05	Template update
*E	2022-11-03	<p>Updates to sync up changes in the code example CE234992 to support MTB 3.0.</p> <p>Clarified the differences between the 1st and 2nd generation PSoC™ 6 devices.</p> <p>Updated several figures to fix errors and sync to CE234992 changes.</p> <p>Several minor changes to fix typos.</p>

5.16 [AN213924 PSoC™ 6 MCU Device Firmware Update \(DFU\) software development kit guide](#)

About this document

-
- 1
- 6

Scope and purpose

This guide provides comprehensive information on how to use the Device Firmware Update (DFU) Software Development Kit (SDK) to develop DFU systems for CYPRESS™ PSoC™ 6 MCU products. A detailed description of the SDK application programming interface (API), and build instructions for multiple bootloader code examples, are included.

5 PSoC™ 6 application notes~~DRAFT~~
5.16.1 Introduction

This guide gives an overview of device firmware update (DFU, also called "bootloader") fundamentals, followed by a detailed description of the CYPRESS™ DFU Software Development Kit (SDK) and how to use it with PSoC™ 6 MCU.

Note: *The term "bootloader" has become overloaded in the industry and is frequently confused with the device startup ("boot-up") process. Therefore, this application note, and the associated code examples are transitioning to "device firmware update", or DFU, to reduce ambiguity.*

Multiple development tools are covered in this guide, including CYPRESS™ ModusToolbox™ integrated development environment (IDE) and CYPRESS™ PSoC™ Creator IDE. ModusToolbox™ IDE is a free IDE for Windows®, macOS®, and Linux®. It provides a single, coherent, and familiar design experience for Internet of Things (IoT) designers, combining PSoC™ 6 MCU and the industry's most deployed WiFi and Bluetooth® technologies. For more information on ModusToolbox™ IDE, click [here](#).

PSoC™ Creator is a free Windows-based IDE that enables concurrent hardware and firmware design of systems based on multiple CYPRESS™ PSoC™ MCU families. For more information on PSoC™ Creator, click [here](#).

If you are new to DFU / bootloaders in general, basic concepts and design principles are explained in the next section, [What Is device firmware update?](#)

If you are familiar with DFU concepts and want to see how they are implemented in PSoC™ 6 MCU, see the sections [DFU SDK description](#), [DFU SDK files](#), and [PSoC™ 6 MCU DFU code examples](#). For a list of the DFU code examples, see [References](#). For a complete list of PSoC™ 6 MCU code examples, click [here](#).

Note: *At the time of publication, UART-, I²C-, SPI-, and BLE-based DFU systems are supported. Other communication channels will be added in future SDK updates*

~~5 PSoC™ 6 application notes~~

~~5.16.2~~ What Is device firmware update?

Device firmware update (DFU) is a common part of MCU system design. DFU makes updating a product's firmware in the field possible. In a typical product, firmware is stored in an MCU's flash memory. The MCU is mounted on a printed circuit board (PCB) and embedded in a product, as [Figure 564](#) shows.

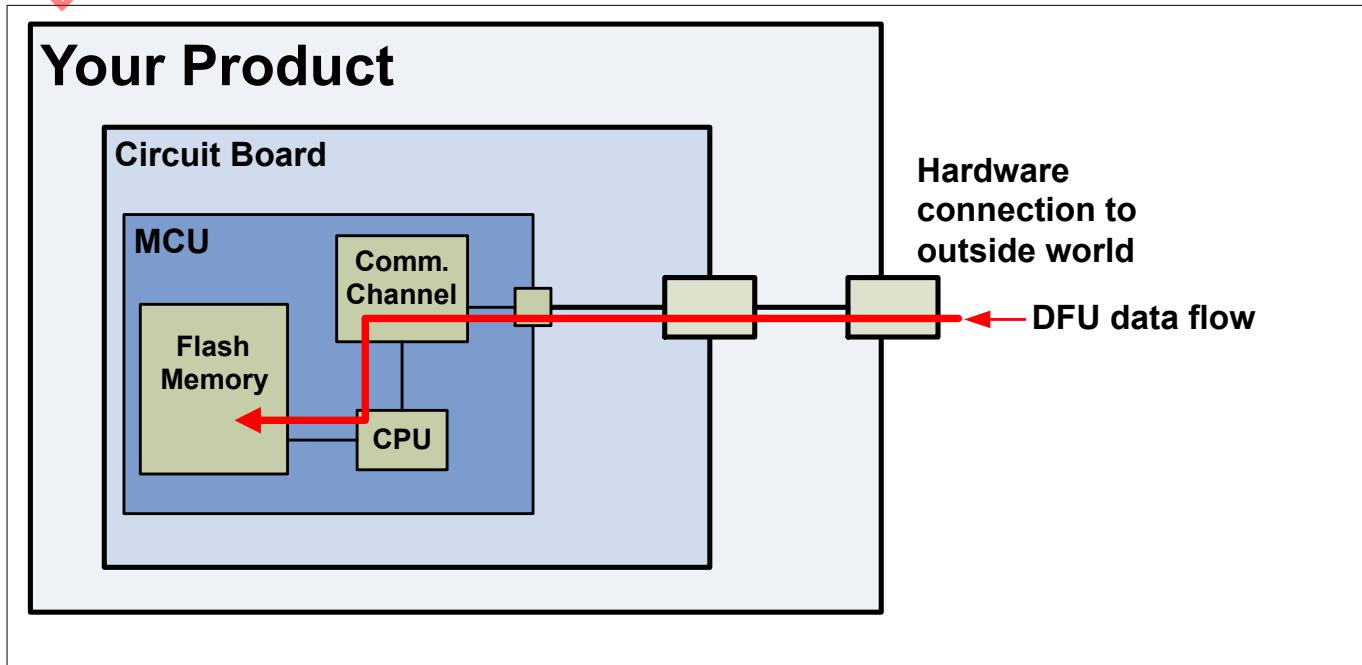


Figure 564 DFU data flow block diagram

At the factory, initial programming of firmware into a product is typically done at PCB assembly time, using the MCU's JTAG or SWD interface. However, these interfaces are not usually available in the field, and therefore are generally not used for firmware updates.

A better way to update firmware in the field is to use an existing connection between the product and the outside world. The connection may be a standard communication port such as I²C, USB, or UART, a wireless channel such as BLE, or a custom protocol.

5.16.2.1 Terms and definitions

[Figure 564](#) implies that the product's embedded firmware must be able to use the communication port for two different purposes: normal operation and updating flash. The portion of the embedded firmware that knows how to update the flash is called a **DFU module**, or **bootloader**, as [Figure 565](#) shows.

5 PSoC™ 6 application notes

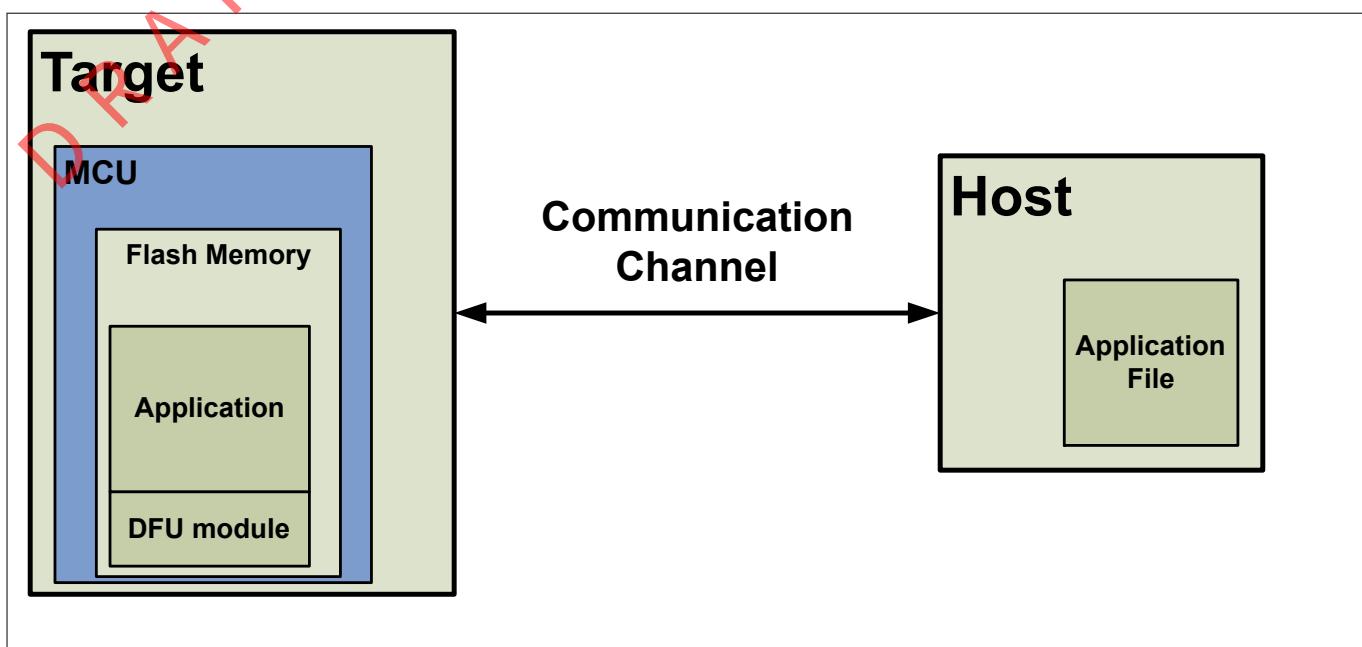


Figure 565 Generic DFU system

Typically, the system that provides the data to update the flash is called the **host**, and the system being updated is called the **target**. The host can be an external computer or another MCU on the same PCB as the target.

The act of transferring data from the host to the target flash is called **DFU**, or a **DFU operation** (In previous documentation, CYPRESS™ referred to the DFU as the bootloader. However, this term is frequently confused with the boot-up process. Therefore, in this application note and associated code examples, we refer to this as DFU to reduce ambiguity.) The data that is placed in flash is called the **application** or **firmware image**.

5.16.2.2 Using a DFU module

The DFU module and the application typically share a communication port. The first step in DFU is to put the product in a mode where the DFU module, and not the application, is executing. This can be done in response to an event such as a button press or a received command. The application detects such an event and responds by transferring control to the DFU module.

After the DFU module is running, the host can send a “start DFU” command over the communication channel. If the DFU module sends an “OK” response, a DFU operation can begin.

5.16.2.3 Basic DFU function flow

During DFU, the host reads the file for the new application, parses it into flash program commands, and sends those commands to the DFU module. After the entire file is received and installed in target flash, the DFU module can pass control to the new application.

A DFU module typically executes first after device reset.³⁹ It can then perform the following actions:

- Check the new application’s validity before transferring control to that application
- Manage the timing to start host communication

³⁹ In a typical MCU, several events may cause a device reset: for example, a device power up or a voltage level on a reset pin. In addition, firmware can trigger a reset by writing to a device register. This is known as a “software reset”, or SRES. Using the SRES feature, an application can reset the device, for example in response to an event such as a button press. This effectively transfers control to the DFU module because the DFU module executes first at device reset.

5 PSoC™ 6 application notes

- ~~DRAFT~~
- Do the DFU operation
 - Pass control to the new application

5.16.2.4 Other use cases

Other more complex DFU use cases exist. A common use case is for an application to be running and have some event alert the application that an update is available.

While the application continues to execute its normal tasks, it also downloads the new application into a temporary location. After the new application is checked for validity, it is copied into the correct location in flash, and control is transferred to it. [Figure 566](#) is a flow diagram that shows how this works.

5 PSoC™ 6 application notes

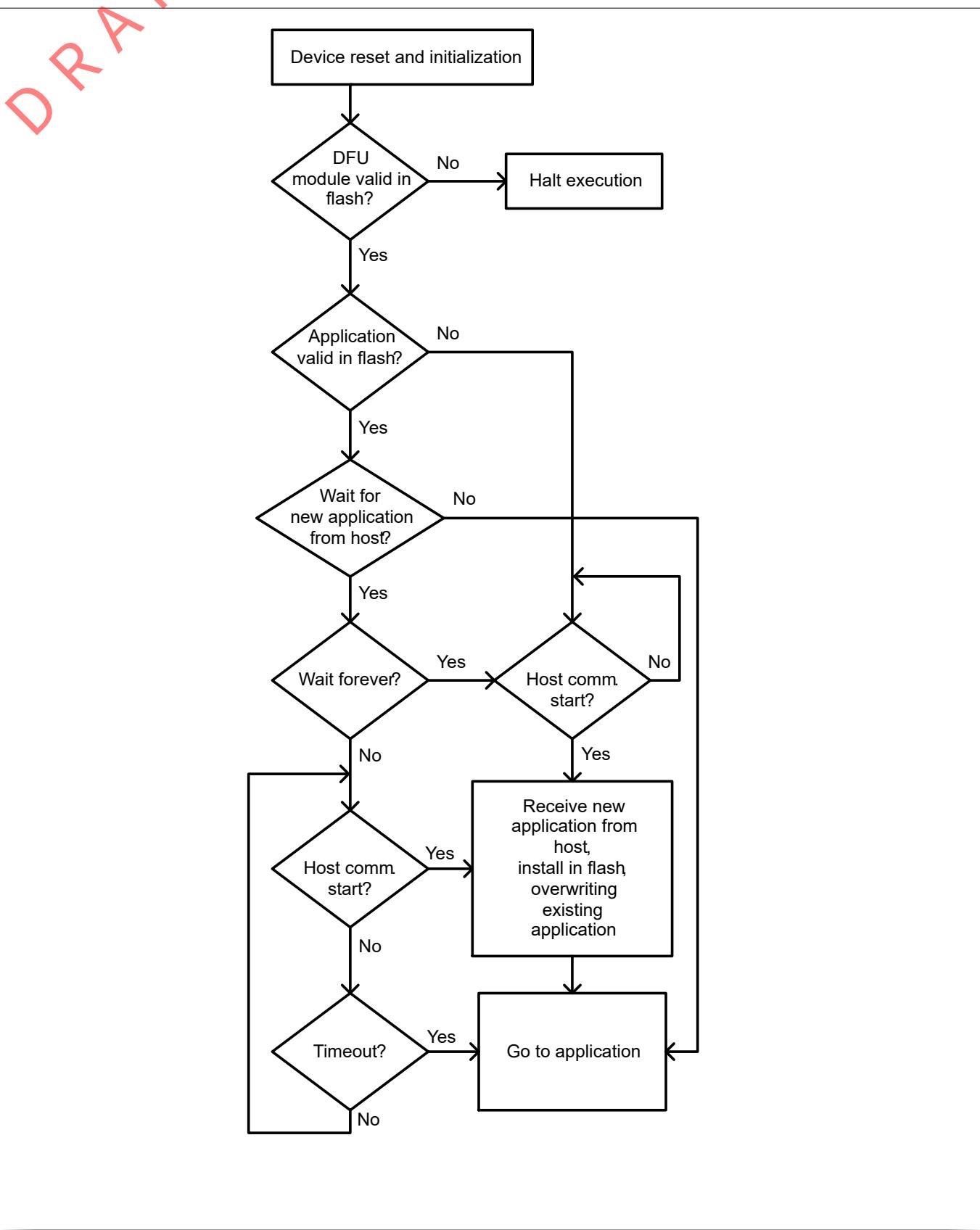


Figure 566

DFU function flow

~~DO NOT USE~~ 5 PSoC™ 6 application notes

5.16.3 DFU SDK description

The CYPRESS™ DFU SDK is an API consisting of a set of callable functions and other elements that enable rapid DFU system development. The SDK is shipped as a part of the CYPRESS™ PSoC™ 6 SDK. The SDK is typically found in C:\...\ModusToolbox_1.0\libraries\PSoC6sw1.0\components\PSoC6mw\dfu. The SDK may be used by other IDEs as well as ModusToolbox IDE.

The SDK consists of the following:

- Source code (typically .h and .c files as well as linker scripts and .mk files) that implements the SDK API
- Documentation; i.e., this guide
- Code examples

Note: *You may modify the DFU SDK source code for custom purposes, for example to modify or add commands to the host interface protocol (see [Appendix B](#)).*

5.16.3.1 DFU SDK files

[Table 137](#) lists the files in the SDK.

Note: *PSoC™ 6 MCU has two Arm® CPUs: a Cortex®-M4 (CM4) and a Cortex-M0+ (CM0+). Linker script files are provided for each CPU, for multiple IDEs. For more information, see [AN215656](#), PSoC™ 6 MCU Dual-CPU System Design.*

Table 137 DFU SDK files

File	Description
cy_dfu.h, .c	The DFU SDK files
cy_dfu_bwc_macro.h	Enables backward compatibility with the Cypress legacy Bootloader SDK
dfu_user.h	Contains user-editable #define statements that control the operation and the enabled features in the SDK
dfu_user.c	Contains user functions required by the SDK: Five functions that control communications with the DFU host; these are also called “transport functions” Two functions – ReadData () and WriteData () – that control access to internal or external memory The functions provided are defaults and may be modified for your application
transport_uart.h, .ctransport_... .h, .c	Contains DFU transport functions for the host communication channel being used; these functions are typically called by the transport functions in dfu_user.c
dfu_cm4.ld, dfu_cm0p.ld	Custom GCC linker scripts
dfu_cm4.scat, dfu_cm0p.scat, dfu_mdk_common.h, dfu_mdk_symbols.c	Custom MDK linker scripts; the common files exist to create necessary symbol definitions.
dfu_cm4.icf, dfu_cm0p.icf	Custom IAR linker scripts

The files are organized in the DFU folder as [Figure 567](#) shows:

5 PSoC™ 6 application notes

```

DRAFT
\---dfu
|   cy_dfu.h
|   cy_dfu.c
|   cy_dfu_bwc_macro.h
\---config
|   dfu_user.h
|   dfu_user.c
|   transport_uart.h
|   transport_uart.c
|   transport_... .h  transport files for other
|   transport_... .c  communication channels
\---linker_scripts
    \---GCC
        |   dfu_cm0p.ld
        |   dfu_cm4.ld
    \---MDK
        |   dfu_cm0p.scat
        |   dfu_cm4.scat
        |   dfu_mdk_common.h
        |   dfu_mdk_symbols.c
    \---IAR
        dfu_cm0p.icf
        dfu_cm4.icf

```

Figure 567 DFU SDK file organization

5.16.3.1.1 User callback functions

The DFU API functions call back to several user functions. This allows you to customize the following DFU operations:

- Host communication, also called transport functions or communication interface functions
- Reading and writing device's internal flash as well as other nonvolatile memory (NVM), for example external flash

Table 138 lists the required user callback functions. The DFU code examples show typical code for these functions; see [PSoC™ 6 MCU DFU code examples](#) and [References](#).

Table 138 User callback functions

Function	Description
For host communication. Examples are in <code>dfu_user.c</code> and <code>transport_xxx.h</code> and <code>.c</code> files	
<code>Cy_DFU_TransportStart()</code>	Opens and initializes the communication channel
<code>Cy_DFU_TransportStop()</code>	Closes the communication channel
<code>Cy_DFU_TransportReset()</code>	Re-initializes the channel, typically to bring it back to a known state
<code>Cy_DFU_TransportRead()</code>	Receives a packet from the host; see Appendix B
<code>Cy_DFU_TransportWrite()</code>	Sends a packet to the host; see Appendix B

(table continues...)

~~5 PSoC™ 6 application notes~~

~~DRAFT~~ **Table 138 (continued) User callback functions**

Function	Description
For non-volatile memory (NVM) access. Examples are in dfu_user.c	
Cy_DFU_ReadData()	Reads data from the device flash or other NVM
Cy_DFU_WriteData()	Writes data to the device flash or other NVM

5.16.3.1.2 Linker scripts

A DFU system is a system with multiple applications (at least two) in NVM. You decide where in NVM each application resides. The DFU SDK includes template linker script files (see [Table 137](#) and [Figure 567](#)) that you can adapt to your application. The DFU code examples include example linker script files.

5.16.3.2 DFU ecosystem

Other elements of the DFU system are not included in the SDK. They are available in ModusToolbox™ IDE, elsewhere in the PSoC™ 6 SDK, in the PSoC™ Creator IDE, or in the peripheral driver library (PDL), and can be used by other IDEs:

- CyMCUElfTool: cymcuelftool.exe in the folder ... \ ModusToolbox_1.0 \ tools \ cymcuelftool1.0 \ bin. This program is called as part of building an application, as [Figure 568](#) shows. (The diagram is similar for PSoC™ Creator and other IDEs.)
- This program combines core and application image files into output files. It is a command line program; its option syntax is documented in the Help command (-h / -help) output. A user guide is available in the folder ... \ cymcuelftool1.0 \ doc.
- Other drivers – for communication, flash, etc. – are in the folder ... \ ModusToolbox_1.0 \ libraries \ PSoC6sw1.0 \ components \ PSoC6pd1 \ drivers. They are called as needed by the .DFU SDK API functions
- Device Firmware Update Host Tool: dfuhtool.exe in the folder ... \ ModusToolbox_1.0 \ tools \ dfuhtool1.0. This is an example of the “host” shown in [Figure 565](#). For more information, see [Appendix A](#).

5 PSoC™ 6 application notes

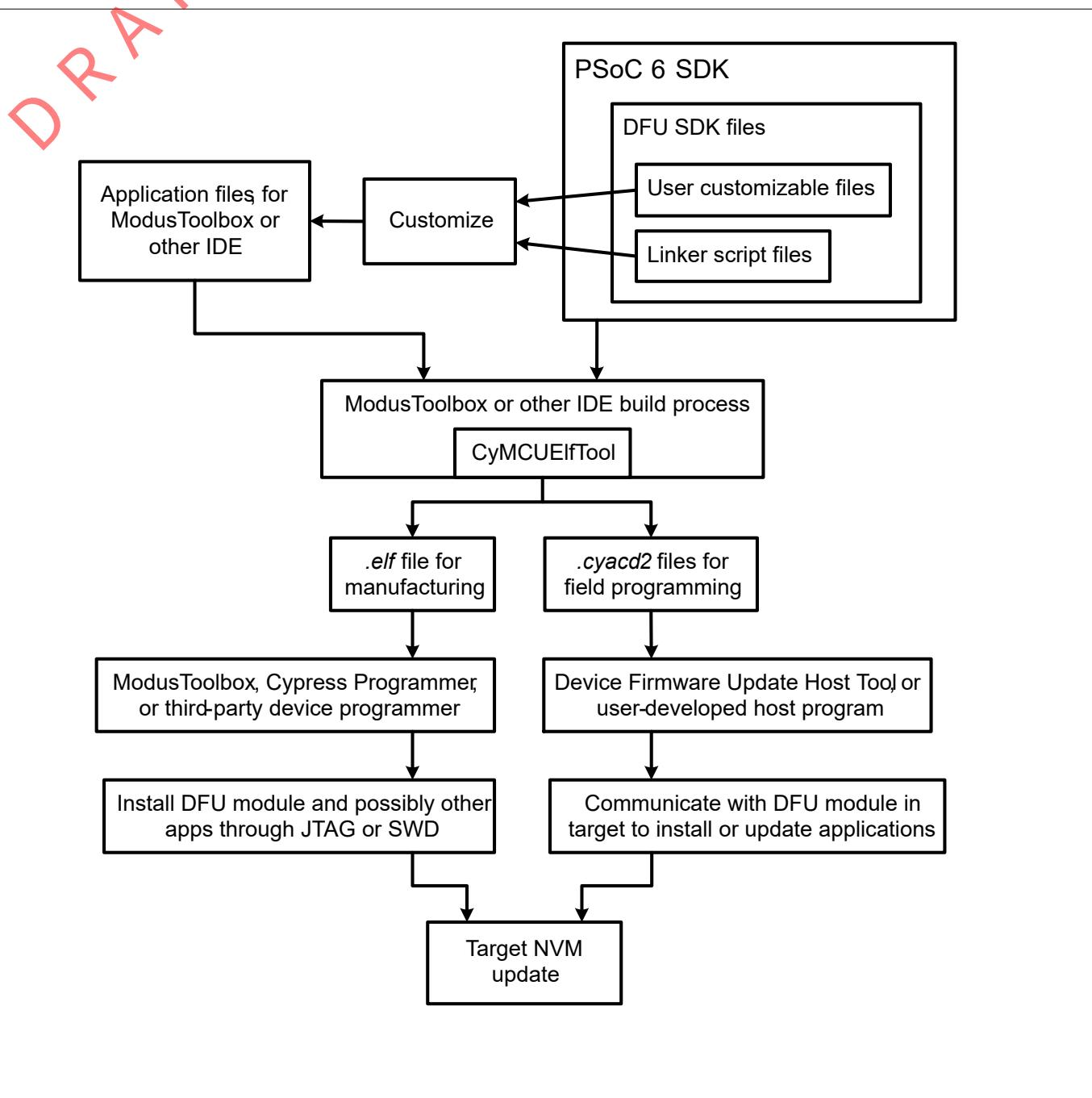


Figure 568 ModusToolbox™ application build and DFU process diagram

~~5 PSoC™ 6 application notes~~

~~5.16.4 How to use the SDK~~

This section describes the general process for using the DFU SDK. At a high level, the steps are Comparison of Devices

- [Determine the applications in your system](#)
- [Locate applications in memory](#)
- [Design the applications](#)
- [Build and program the applications](#)

For detailed instructions on how to build the DFU SDK code examples, go to [PSoC™ 6 MCU DFU code examples](#).

5.16.4.1 Determine the applications in your system

The first step is to decide how many applications are to be in your design, and the purpose of each. Which applications will have DFU capability? [Figure 569](#) shows some typical memory maps for multiple applications:

- Map **A** shows a basic case, where Application #0 downloads and installs Application #1
- Map **B** shows a more complex case. There are three applications. All applications have DFU modules – they can download and install any other application. (In practice, Application #0 is usually not updated.) A general data section is located outside all applications
- Map **C** shows the same case as map **B** except that one of the applications is located in external nonvolatile memory (NVM)

In each of the maps, the DFU system uses a small amount of flash (typically one row) to store application metadata. For information, see [Appendix D](#).

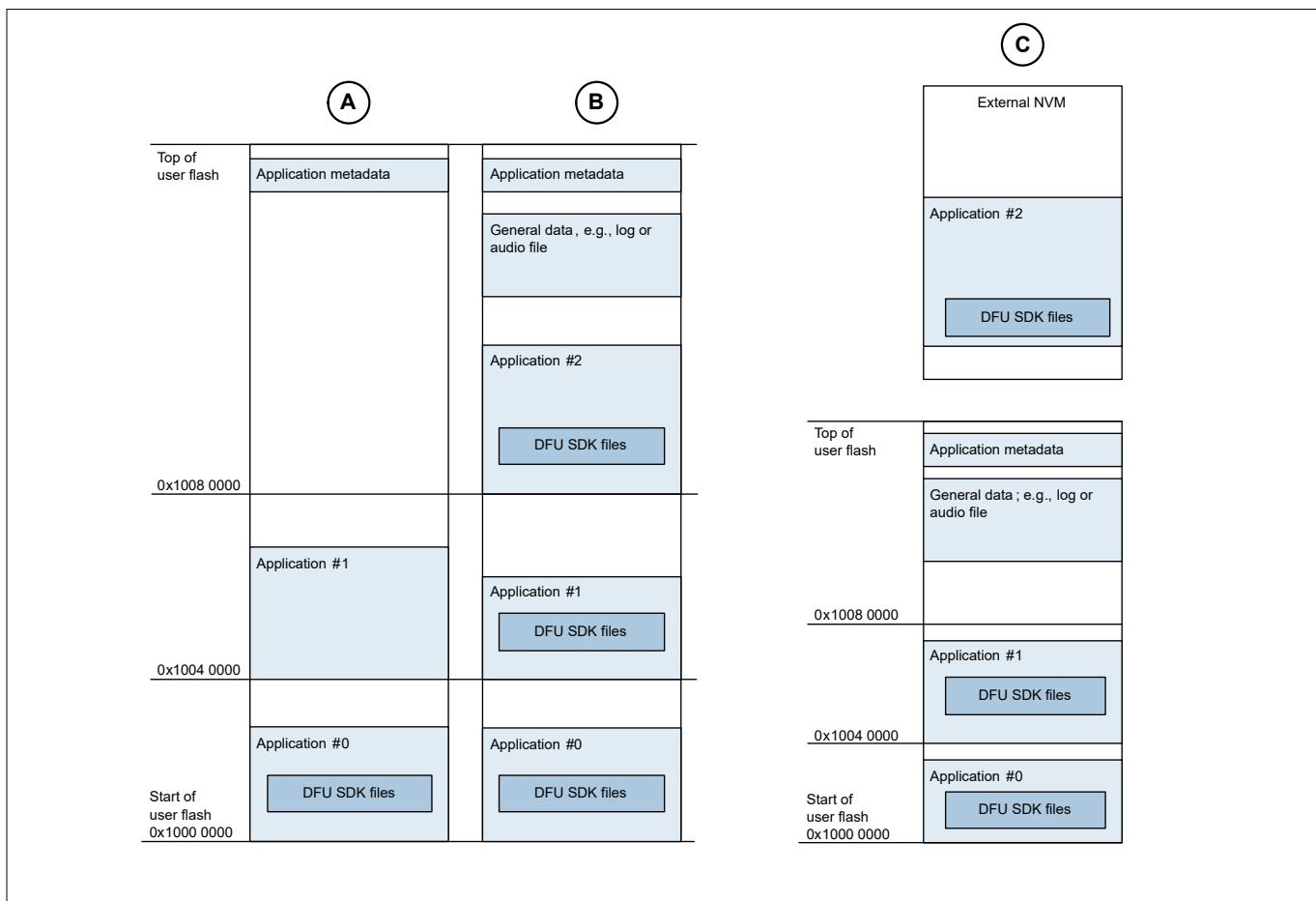


Figure 569

Example DFU memory maps

~~DEAF~~ 5 PSoC™ 6 application notes

5.16.4.2 Locate applications in memory

The next step is to decide where each application is to be located in flash. There are several factors to consider:

- A maximum of 63 applications are supported, depending on the metadata size. For more information on DFU SDK metadata, see [Appendix D](#).
- An application may have code and data for one or more CPUs, for example CM4 and CM0+ in a PSoC™ 6 MCU device. For more information, see the application note [AN215656](#), PSoC™ 6 MCU Dual-CPU System Design
- The first application, usually called “App0”, must be located at the beginning of user flash, which in PSoC™ 6 MCU is at address 0x1000 0000, as [Figure 569](#) shows. This is the first application to execute after device reset⁴⁰⁾.

Note: *Another module, such as a Secure Image module in a secure system, may occupy address 0x1000 0000, and App0 may be located elsewhere in flash. For more information, see [AN221111](#), Creating a Secure System*

- Any application may incorporate the DFU SDK, and so be able to download and install another application, and/or transfer control to another application. In most cases, App0 should incorporate the DFU SDK
- When writing flash in non-blocking mode, PSoC™ 6 MCU has a read-while-write (RWW) capability across 512-KB regions. That is, the DFU code may execute in one flash region while updating flash in another region. For more information, see the PDL flash driver documentation and the device [Technical Reference Manual \(TRM\)](#).
- Applications may reside in external memory as well as the device’s internal flash, as map **C** in [Figure 569](#) shows. External memory is in a 128-MB sub-region from 0x1800 0000 to 0x1FFF FFFF

Note: *When executing out of external memory, the external memory device must be mapped into the PSoC™ 6 MCU using the QSPI block in XIP mode. For more information, see the device [Technical Reference Manual \(TRM\)](#).*

- PSoC™ 6 MCU applications include validation bytes, which are saved in NVM, as part of the application. This enables an application to be validated before transferring control to it

Note: *Depending on the security environment within which the applications exist, an application can be structured for simple validation (with checksum or CRC data) as well as authorization (with encrypted signature data). The DFU system can either directly implement validation or call a function in the Flash Boot module to check authorization. For more information, see [AN221111](#), Creating a Secure System*

- NVM data that may be updated during the normal workflow, for example an events log, should not be part of an application, because it makes the application difficult to validate. Instead, designate a region of NVM that is outside any application’s space and use it for data storage. [Figure 569](#) shows two examples in maps **B** and **C**
- Reserve one row of flash for application metadata, as [Figure 569](#) shows. (In rare cases more than one row may be required.) Application metadata is managed by the DFU SDK and is used by applications to transfer control to another application. Application metadata space should be outside any application space

⁴⁰ The CM0+ CPU executes SRAM and other system-level code first at device reset, then transfers control to the application residing at 0x1000 0000. For more information, see the [Technical Reference Manual \(TRM\)](#).

5 PSoC™ 6 application notes

5.16.4.3 Design the applications

5.16.4.3.1 ModusToolbox™ instructions

Note: This section contains only DFU-specific instructions. For detailed step-by-step instructions for creating a ModusToolbox™ application, see [ModusToolbox™ Help](#); [AN221774, Getting Started with PSoC™ 6 MCU](#); or [PSoC™ 6 MCU DFU code examples](#) in this document.

The next step is to design each of the applications identified in the [previous steps](#). Each application is a single ModusToolbox™ application, independent from any other application. With ModusToolbox™ IDE, you can have all the applications in one workspace, or in separate workspaces, as well as in separate locations on your computer. Before getting started with PSoC™ 6 MCU, developing a plan for workspaces and applications for your overall system development needs is recommended.

Note: A ModusToolbox™ application for PSoC™ 6 MCU (dual-CPU) is composed of five projects:

- <project name>_config: the device configuration file design.modus, and device configuration source files
- <project name>_mainapp: source files for the Cortex-M4 (CM4) CPU
- <project name>_mainapp_cm0p: source files for the Cortex-M0+ (CM0+) CPU
- <project name>_mainapp_psoc6pdl: driver files for the Cortex-M4 (CM4) CPU
- <project name>_mainapp_cm0p_psoc6pdl: driver files for the Cortex-M0+ (CM0+) CPU

Do the following for each ModusToolbox™ application that is to do DFU operations and/or be an installable application:

1. If the application is to do DFU operations, enable a communication peripheral in the design.modus file. That peripheral implements the communication channel to the DFU host
The DFU SDK includes support for many communication channel types, including UART, SPI, I²C, and BLE. Code examples are available for each channel type; see [PSoC™ 6 MCU DFU code examples](#) and [References](#)
You can also create a custom communication channel. The driver must implement the transport functions described in [User callback functions](#).
2. Incorporate the DFU SDK into the project, as [Figure 570](#) shows. Right-click the _mainapp project in the Project Explorer window and click **Middleware Selector**.

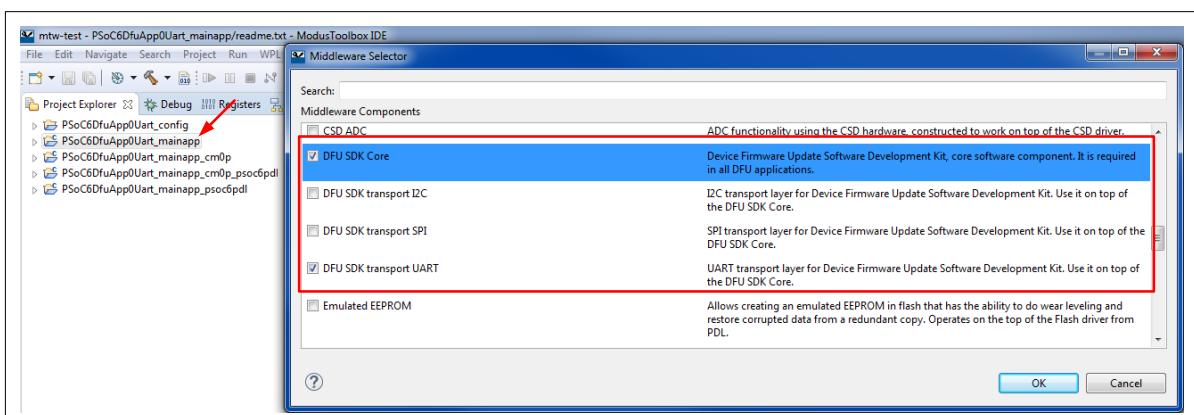


Figure 570 Incorporate DFU SDK into a ModusToolbox™ Project

If the application will do DFU operations, select **DFU SDK Core** and a **DFU SDK transport xxx** box

5 PSoC™ 6 application notes

DRAFT

Note: In most cases, only one transport box is selected. However, if you want to have a custom communication channel, you can leave all transport boxes unselected, or even select multiple boxes.

If the application is a downloadable application and will transfer control to another application, select **DFU SDK Core**. The code to transfer control to another application is included in the DFU SDK core files Click **OK** when done. Required source .c, .h, and linker script (e.g., .ld) files are automatically added to the project

Repeat this step for the _mainapp_cm0p project; but select only the **DFU SDK core** box

3. Installed linker script files are by default set up for Application #0 (App0). For other applications, edit the files by changing the application number. The following example shows edits for downloadable application App1, for the GCC linker, in dfu_cm0p.ld and dfu_cm4.ld:

```
/*
 * DFU SDK specific: aliases regions, so the rest of the code does not use
 * application-specific memory region names
 */
REGION_ALIAS("flash_core0", flash_app1_core0);
REGION_ALIAS("flash", flash_app1_core1);
REGION_ALIAS("ram", ram_app1_core1);

/* DFU SDK specific: sets an app Id */
__cy_app_id = 1;
```

4. Change the project properties to use the DFU linker script files instead of the default linker script files. Select the dfu_cm4.ld file for _mainapp, as [Figure 571](#) shows. Include the relative path to the file. Select the dfu_cm0p.ld file for _mainapp_cm0p (not shown in the figure)

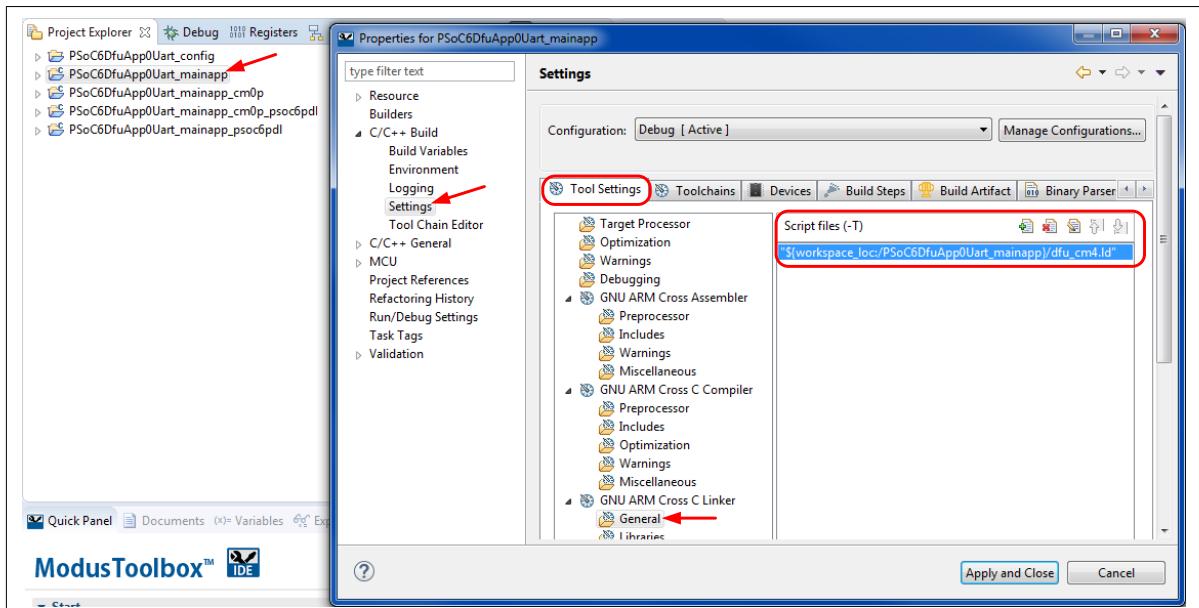


Figure 571 ModusToolbox™ project properties setting for custom linker script

5. For downloadable application projects, add a post-build command to call the CYPRESS™ utility program cymcuelftool.exe, as [Figure 572](#) shows. **cymcuelftool** is included with your ModusToolbox™ installation. It generates a *.cyacd2 file, which is downloaded by the DFU host (see [Figure 568](#)). The post-build command is done only for the CM4 binary. cymcuelftool combines the CM4 (_mainapp) and CM0+ (mainapp_cm0p) projects to create the output .cyacd2 file

5 PSoC™ 6 application notes

DRAFT

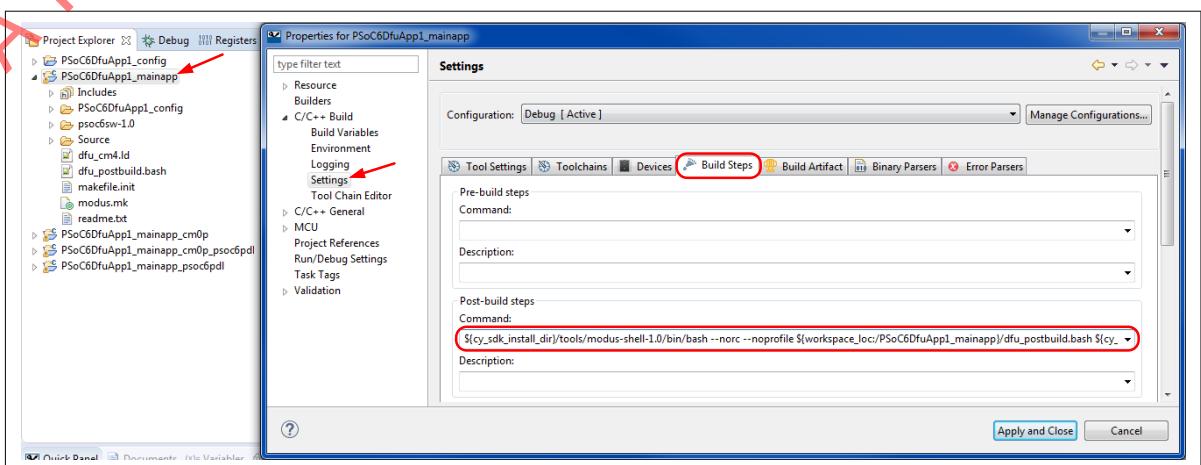


Figure 572

ModusToolbox™ post-build command

The command typically invokes a bash file such as dfu_postbuild.bash in code example [CE213903](#), PSoC™ 6 MCU Basic DFU. For convenience, copy and paste the file into your _mainapp project

Then change the _mainapp project properties to add command text similar to the following, as [Figure 572](#) shows

```

${cy_sdk_install_dir}/tools/modus-shell-1.0/bin/bash --norc --noprofile
${workspace_loc:/PSoC6DfuApp1_mainapp}/dfu_postbuild.bash
${cy_sdk_install_dir}/tools/cymcuelftool-1.0/bin/cymcuelftool
${workspace_loc:/PSoC6DfuApp1_mainapp_cm0p}/${config_name:PSoC6DfuApp1_mainapp_cm0p}/
PSoC6DfuApp1_mainapp_cm0p.elf
${workspace_loc:/PSoC6DfuApp1_mainapp}/${config_name:PSoC6DfuApp1_mainapp}/
PSoC6DfuApp1_mainapp.elf ARM_CM4

```

6. Add code as needed to the `main.c` and other source files in both the CM4 (_mainapp) and CM0+ (mainapp_cm0p) projects. Specific requirements are:

- Add a `#include "cy_dfu.h"` statement to all source files as needed to access the DFU SDK API
- For downloadable applications, in `main.c` in the `_mainapp_cm0p` project, change the `Cy_SysEnableCM4()` function call to: `Cy_SysEnableCM4((uint32_t)(&__cy_app_core1_start_addr))`;
- For downloadable applications, in `main.c` in the `_mainapp` project, add the following global statement. Adjust the array size for the selected signature type

```

/* This section holds signature data for application verification. */
CY_SECTION(".cy_app_signature") __USED static const uint32_t
cy_bootload_appSignature[1];

```

In your overall code in both main files, consider the following:

- Which DFU function(s) to call:
 - `Cy_DFU_Complete()`: blocks while doing the entire DFU operation. Call this function if there is no other task to do during DFU
 - `Cy_DFU_Init()` followed by a series of calls to `Cy_DFU_Continue()`: `Cy_DFU_Continue()` blocks while receiving, processing, and responding to one command packet from the host. Call these functions if other tasks must be done during DFU
- Whether each application shall pass control to another application. Add calls to DFU SDK API functions `Cy_DFU_ValidateApp()` and `Cy_DFU_ExecuteApp()` as needed

5 PSoC™ 6 application notes

For more information, see the PDL DFU SDK documentation.

Note: You can copy and paste code from the code examples listed in [PSoC™ 6 MCU DFU code examples](#) and [References](#).

5.16.4.3.2 PSoC™ Creator instructions

Note: This section contains only bootloader-specific (in ModusToolbox™ IDE, this has been changed to DFU operation) instructions. For detailed step-by-step instructions for creating a PSoC™ Creator project, see [PSoC™ Creator Help](#); [AN210781](#), Getting Started with PSoC™ 6 MCU with Bluetooth Low Energy (BLE) Connectivity; or [How to build the code examples](#) in this document.

The next step is to design each of the applications identified in the [previous steps](#). Each application is a single PSoC™ Creator project, independent from any other project. With PSoC™ Creator, you can have all projects in one workspace (.cywrk file), or in separate workspaces as well as in separate locations on your computer. Before getting started with PSoC™ 6 MCU, developing a plan for workspaces and projects for your overall system development needs is recommended.

Do the following for each PSoC™ Creator project that is to be a bootloader or an installable application:

1. If the application is to do bootloading, place a communication Component on the project schematic; i.e., the TopDesign.cysch file. This Component implements the communication channel to the bootloader host. Connect the Component terminals to the appropriate physical pins.

The Bootloader SDK includes support for many of the communication Components in the PSoC™ Creator Component Catalog, including UART, SPI, I²C, and BLE. Code examples are available for each communication Component; see [PSoC™ 6 MCU BLE bootloaders](#) and [References](#).

Note: For compatibility with the default transport files in the SDK, name the Component as **UART**, **I2C**, **SPI**, or **BLE**

You can also create a custom communication channel. The driver must implement the transport functions described in [User callback functions](#)

2. Incorporate the Bootloader SDK into the project, as [Figure 573](#) shows. Right-click the project in the Workspace Explorer window and select **Build Settings...**. In the **Build Settings** dialog, select **Peripheral Driver Library**.

5 PSoC™ 6 application notes

DRAFT

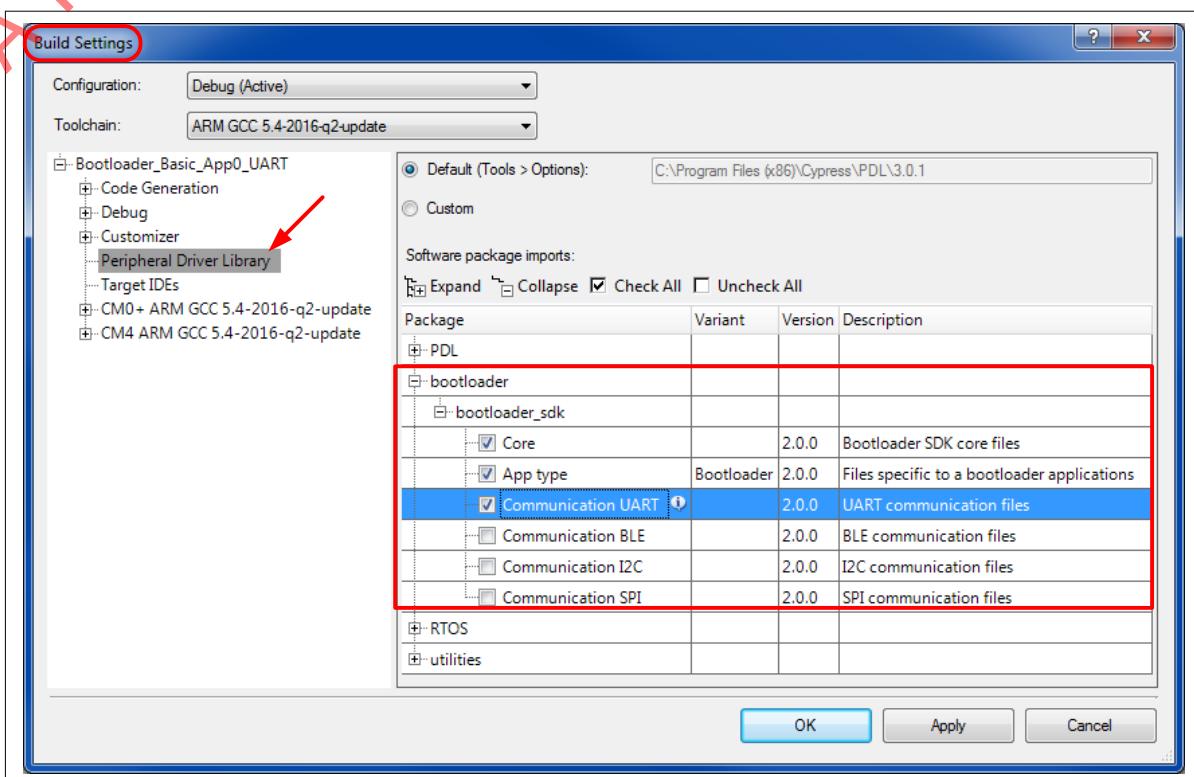


Figure 573 Incorporate bootloader SDK into a PSoC™ Creator project

If the application will do bootload operations, select **Core**, **App type bootloader**, and a **Communication** box

Note: *In most cases, only one communication box is selected. However, if you want to have a custom communication channel, you can leave all communication boxes unselected, or even select multiple boxes.*

If the application is a downloadable application and will transfer control to another application, select **Core**. The code to transfer control to another application is included in the SDK core files

Click **OK** when done. Required source .c, .h, and linker script (e.g., .ld) files are automatically added to the project

3. Generate the project files. Click the project in the Workspace Explorer window and select **Build > Generate Application**. The files are added to the project, as [Figure 574](#) shows.

Note that PSoC™ 6 MCU has two Arm CPUs: CM4 and CM0+. Each CPU has its own folder, plus Generic folders and a shared files folder. Linker script files to locate the code for each CPU, for multiple IDEs, are automatically copied and added to the project. For more information, see [AN215656](#), PSoC™ 6 MCU Dual-CPU System Design.

Note: [Figure 574](#) shows bootloader files added to the CM0p folder. Similar files are also added to the CM4 folder but are not shown

5 PSoC™ 6 application notes

DRAFT

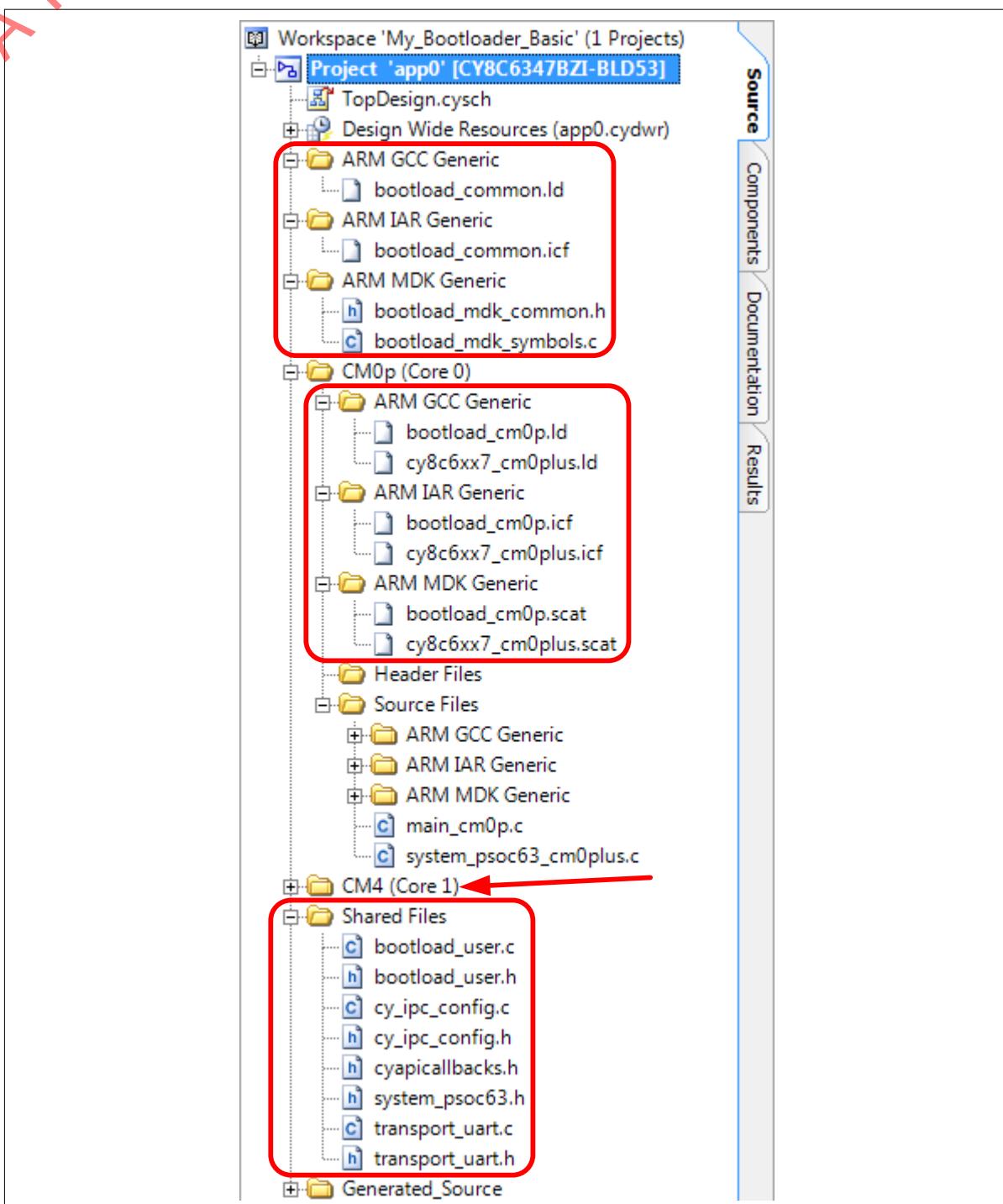


Figure 574 Add SDK files to a PSoC™ Creator project

4. Edit the files `bootload_user.h` and `bootload_user.c`. Review the [User callback functions](#) in `bootload_user.c`, and edit them for any customization that is needed, for example, to write and read external memory. Usually the functions can be left unchanged

Note: The `bootload_user.c` file that is added contains the default code for UART. For other communication channels such as I²C and SPI, edit `bootload_user.c` as follows:

- Change: `#include "transport_uart.h"` to: `#include "transport_i2c.h"` or: `#include "transport_spi.h"` or: `#include "transport_ble.h"`
- Change five instances of “UART_Uart” to “I2C_I2c”, “SPI_Spi”, or “CyBLE”.

5 PSoC™ 6 application notes

- ~~DRAFT~~
5. Depending on the compiler you are using, edit the appropriate common linker script file, to encode the decisions made in [Locate applications in memory](#). For an example, see [Section 5.3 Step 9](#)
 6. Installed linker script files are by default set up for application #0 (app0). For other applications, edit the files by changing the application number. The following example shows edits for app1, for the GCC linker, in `bootload_cm0p.ld` and `bootload_cm4.ld`:

```

/*
 * Bootloader SDK-specific: aliases regions, so the rest of the code does not use
 * application-specific memory region names*/

REGION_ALIAS("flash_core0", flash_app1_core0);
REGION_ALIAS("flash",     flash_app1_core1);
REGION_ALIAS("ram",      ram_app1_core1);

/* Bootloader SDK-specific: sets App ID */
__cy_app_id = 1;

```

The Keil µVision MDK linker is more complex. The following example shows edits for app1 in `bootload_cm0p.scat` and `bootload_cm4.scat`:

```

; Flash
#define FLASH_START          CY_APP1_CORE0_FLASH_ADDR
#define FLASH_SIZE           CY_APP1_CORE0_FLASH_LENGTH

; Emulated EEPROM Flash area
#define EM_EEPROM_START      CY_APP1_CORE0_EM_EEPROM_ADDR
#define EM_EEPROM_SIZE        CY_APP1_CORE0_EM_EEPROM_LENGTH

; External memory
#define XIP_START             CY_APP1_CORE0_SMIF_ADDR
#define XIP_SIZE               CY_APP1_CORE0_SMIF_LENGTH

; RAM
#define RAM_START              CY_APP1_CORE0_RAM_ADDR
#define RAM_SIZE                CY_APP1_CORE0_RAM_LENGTH

```

And edits for App1 in `bootload_mdk_symbols.c`:

```

__cy_app_core1_start_addr    EQU __cpp(CY_APP1_CORE1_FLASH_ADDR)

/* Application number (ID) */
__cy_app_id                  EQU 1

/* CyMCUElfTool uses these to generate an application signature */
__cy_app_verify_start        EQU __cpp(CY_APP1_CORE0_FLASH_ADDR)
__cy_app_verify_length       EQU __cpp(CY_APP1_CORE0_FLASH_LENGTH +
                                      CY_APP1_CORE1_FLASH_LENGTH -
                                      __CY_BOOT_SIGNATURE_SIZE)

```

5 PSoC™ 6 application notes

- ~~DRAFT~~
7. Change the Project Build settings to use the bootloader linker script files instead of the default linker script files. Select the `bootload_cm0p.ld` file for the CM0+ CPU, as Figure 575 shows. Include the relative path to the file. Select the `bootload_cm4.ld` file for the CM4 CPU (not shown in the figure)

~~DRAFT~~
Note: *Linker script files are provided for GCC, Keil µVision MDK, and IAR Embedded Workbench linkers. The example shown is for GCC. Use the appropriate linker script for your IDE.*

Note: *This operation should be done for both Debug and Release configurations.*

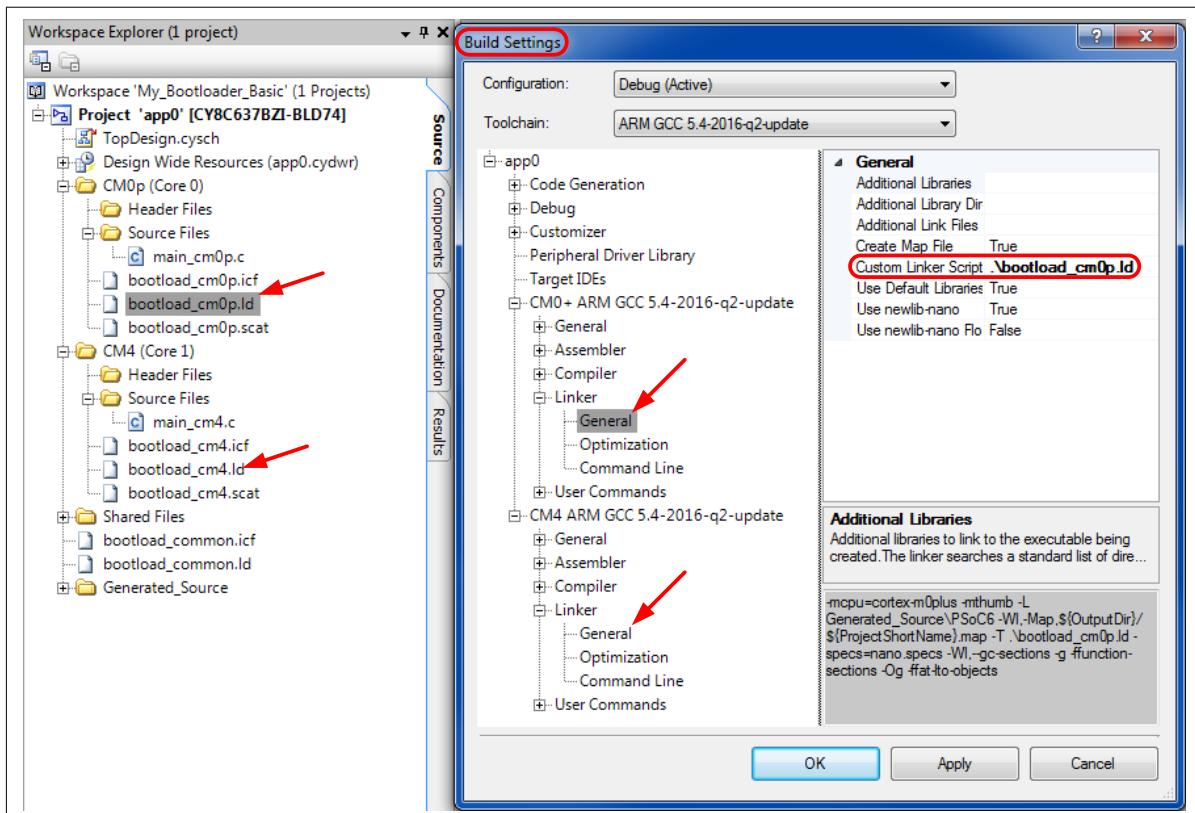


Figure 575 PSoC Creator project build settings for custom linker scripts

8. For downloadable application projects, add a post-build batch file to call the CYPRESS™ utility program `cymcuelftool.exe`. `cymcuelftool` is included with your PSoC™ Creator installation. It generates a `*.cyacd2` file, which is downloaded by the bootloader host (see Figure 568). The batch file is applied only to the CM4 binary.

Note: See a Bootloader SDK code example such as [CE213903, PSoC™ 6 MCU Basic Bootloaders](#), for an example of a batch file. It is usually found in folder `CE213903 | Bootloader_Basic_App1.cydsn`. The batch file contents are also available in [Appendix F](#). For convenience, copy and paste the file into your PSoC™ Creator project folder `app1.cydsn`

Then in PSoC Creator, in the Workspace Explorer window, add the batch file to the project's Shared Files folder, and add that file as a post-build command to the Cortex-M4 build, as Figure 576 shows

Note: *This operation must be done for both Debug and Release configurations*

5 PSoC™ 6 application notes

DRAFT

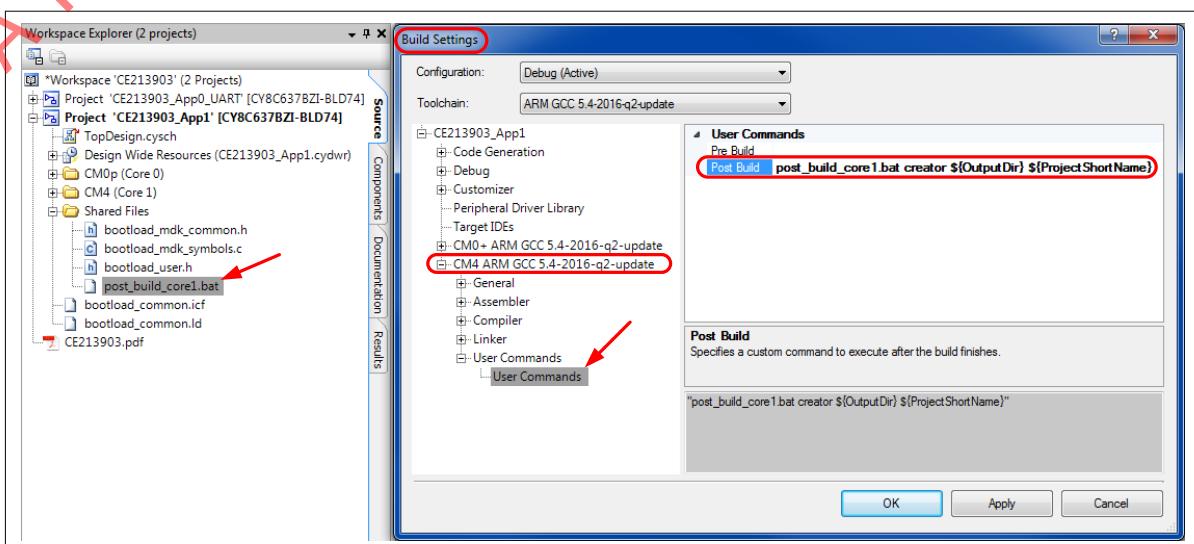


Figure 576 PSoC™ Creator post-build command

Add code as needed to the two `main_cmx.c` and other source files. For downloadbale application projects, add to the `main_cm4.c` file a global array to store the application signature. For example, for checksum, the simplest and most common signature type, add the following:

```
/*
 * This section holds signature data for application verification.
 * For checksum verification, set the number of elements in the array to 1, and
 * in bootload_common.ld set __cy_boot_signature_size = 4.
 */

CY_SECTION(".cy_app_signature") __USED static const uint32_t
    cy_bootload_appSignature[1];
```

In your overall code in both main files, consider the following:

- Which bootloader function(s) to call:
 - `Cy_Bootload_DoBootload()`: blocks while doing the entire bootload operation. Call this function if there is no other task to do while bootloading
 - `Cy_Bootload_Init()` followed by a series of calls to `Cy_Bootload_Continue()`: `Cy_Bootload_Continue()` blocks while receiving, processing, and responding to one command packet from the host. Call these functions if other tasks must be done while bootloading
- Whether each application shall pass control to another application. Add calls to Bootloader SDK API functions `ValidateApplication()` and `ExecuteApplication()` as needed

For more information, see the PDL Bootloader documentation

Note: You can copy and paste code from the code examples listed in [PSoC™ 6 MCU BLE bootloaders](#) and [References](#).

5.16.4.4 Build and program the applications

Build all the applications. Several output files are created, as [Table 139](#) shows for ModusToolbox™ IDE. PSoC™ Creator output files are similar. (See also [Figure 568](#)). All output files are in the Debug or Release folder of their respective projects.

~~DRAFT~~

5 PSoC™ 6 application notes

Table 139 DFU output files for ModusToolbox™ IDE (Example from CE213903, PSoC™ 6 basic DFU)

File name	Description
App0 with DFU module, for ModusToolbox™ IDE	
PSoC6DfuApp0xxx_mainapp_cm0p.elf	Linker output, CM0+ project
PSoC6DfuApp0xxx_mainapp_cm0p_signed.elf	cymcuelftool output after the signing process, CM0+ project
PSoC6DfuApp0xxx_mainapp.elf	Linker output, CM4 project
PSoC6DfuApp0xxx_mainapp_signed.elf	cymcuelftool output after the signing process, CM4 project
PSoC6DfuApp0xxx_mainapp_final.elf	cymcuelftool final output, combining CM4 and CM0+ outputs. This is the file that is programmed to the target device.

App1, the downloadable application, for ModusToolbox™ IDE

PSoC6DfuApp1_mainapp_cm0p.elf	Linker output, CM0+ project
PSoC6DfuApp1_mainapp_cm0p_signed.elf	cymcuelftool output after the signing process, CM0+ project
PSoC6DfuApp1_mainapp.elf	Linker output, CM4 project
PSoC6DfuApp1_mainapp_signed.elf	cymcuelftool output after the signing process, CM4 project
PSoC6DfuApp1_mainapp_final.elf	cymcuelftool final output, combining CM4 and CM0+ outputs.
PSoC6DfuApp1_mainapp_dfu.elf	cymcuelftool final output, signed with CRC bytes. See <code>dfu_postbuild.bash</code> .
PSoC6DfuApp1_mainapp_dfu.cyacd2	cymcuelftool downloadable output. See <code>dfu_postbuild.bash</code> . This is the input to the Device Firmware Update Host Tool.

Program the `App0..._mainapp_final.elf` file into the device, using either ModusToolbox™ IDE or PSoC™ Creator. For more information on device programming, see the Help menus in these tools.

At a later time, you can use the DFU module in Application #0, along with a host program, to download and install application. cyacd2 files into the device. See [Appendix A](#).

~~5 PSoC™ 6 application notes~~

~~5.16.5 PSoC™ 6 MCU DFU code examples~~

There are several code examples associated with this application note. They demonstrate the different ways that [DFU operations](#) can be done.

A complete list of code examples and other documents, with download links on www.cypress.com, is available in [References](#).

[Table 140](#) shows an overview-level list of the code examples:

Table 140 List of PSoC™ 6 MCU DFU code examples

CE #	Title	Description
Supports ModusToolbox™ IDE and PSoC™ Creator		
CE213903	PSoC 6 MCU DFU Basic	<p>A set of examples that demonstrate basic DFU operations; see DFU operations, memory map A:</p> <p>Downloading an application from a host, using various communication channels: UART, I2C, and SPI</p> <p>Installing the downloaded application into user flash</p> <p>Validating an application, and then transferring control to that application</p>
Supports PSoC™ Creator only		
CE216767	PSoC 6 MCU with Bluetooth® Low Energy (BLE) Connectivity Bootloader	Same as CE213903, but uses BLE as the communication channel
CE220959	PSoC™ 6 MCU BLE Bootloader with External Memory	Same as CE216767, but saves the application temporarily in external memory, then copies it to its final destination in user flash
CE220960	PSoC™ 6 MCU BLE Bootloader with Upgradeable Stack	Same as CE216767; in addition the BLE stack can be upgraded
CE221984	PSoC™ 6 MCU Dual-Application Bootloader	Same as CE213903, but demonstrates bootloading two applications, with a factory default (“golden image”) mode; the communication channel is I2C
CE222802	Bootloader with Encryption and Signing	Same as CE213903, but the application is encrypted and signed for validation; the bootloader decrypts the application and validates its signature; the communication channel is UART

The following code examples are planned or are in development:

CE2xxxxx	USB HID DFU	Same as CE213903, but uses USB HID as the communication channel
CE2xxxxx	USB Mass Storage DFU	Same as CE213903, but uses USB Mass Storage as the communication channel
CE2xxxxx	DFU with Multiple Applications	Same as CE221984; each application can update another application; see Figure 569 , memory map B
CE2xxxxx	PSoC™ 6 MCU DFU with External Memory Source	Same as CE213903, but the DFU source is external memory
CE2xxxxx	Embedded Host Program	Similar to that described in AN60317 , PSoC™ 3 and PSoC™ 5LP I2C Bootloader

5 PSoC™ 6 application notes

Most of the code examples consist of multiple separate applications, called “App0” and “App1”. In some cases, additional applications, called “App2”, “App3”, and so on, are included. Each application is a separate application in ModusToolbox™ IDE or a separate project in PSoC™ Creator.

Generally, all applications are in the same ModusToolbox™ or PSoC™ Creator workspace. As noted in [Design the applications](#), projects can exist in separate workspaces as well as in separate locations on your computer.

Usually App0 does the DFU operation; it downloads and installs the other applications. One notable exception is [CE220960](#), BLE Bootloader with Upgradeable Stack – in that code example App1 downloads and installs App2; App0 just transfers control to one of the other applications.

The basic code example, [CE213903](#), has multiple '_App0_ ...' projects; each project has a different communication channel. Advanced communication channels such as BLE and USB are demonstrated in other code examples.

The code examples support the [CY8CKIT-062-BLE PSoC™ 6 BLE Pioneer Kit](#). Applications typically blink a kit LED at different rates or in different colors, making it easy to tell which application is currently running. Porting a code example to another kit or to your system is a straightforward task.

In general, the applications are designed such that you can transfer control from one application to another, by holding a kit button down for 0.5 second. The BLE bootloader code examples use the Immediate Alert Service (IAS) to transfer control between applications.

5.16.5.1 How to build the code examples

The following are step-by-step instructions showing how to build each DFU code example listed in [Table 140](#), with optional adaptations for your application. For more information, see [How to use the SDK](#).

Note: When building the code examples, in many cases it is easiest to copy portions of the existing code example into your project. The copied portions can then be modified for your application. The portions to be copied are listed in each set of instructions.

The instructions are based on Infineon ModusToolbox™ and PSoC™ Creator IDEs; they can be adapted for other IDEs.

5.16.5.2 PSoC™ 6 MCU basic DFUs

This section shows how to build the basic DFUs in code example [CE213903](#). There are five general steps:

- Create App0 and the workspace
- Configure App0 as a DFU
- Add App1
- Configure App1 as an installable application
- Build and test the DFU and the application

These steps apply to both ModusToolbox IDE and PSoC™ Creator:

5.16.5.2.1 ModusToolbox™ instructions

1. Create App0 and the workspace

Start ModusToolbox™ IDE, and select or create the workspace for your applications, as [Figure 577](#) shows:

5 PSoC™ 6 application notes

DRAFT

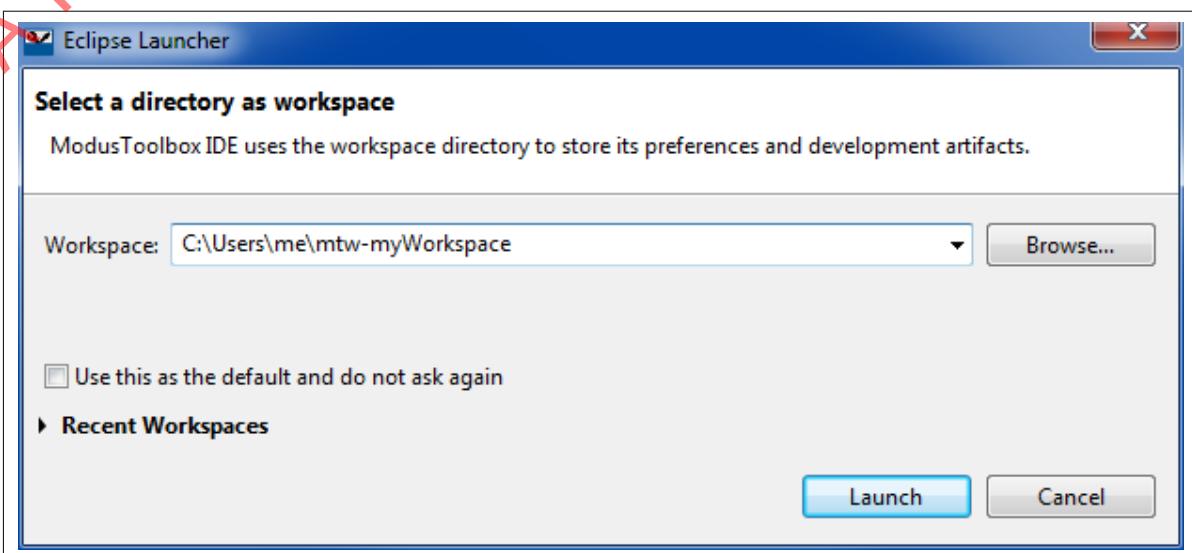


Figure 577 Define a ModusToolbox™ workspace

Create a new application. The easiest way to do this is to click **New Application** in the **Quick Panel** window. In the next dialog, click **Dev/Eval Kit**, and select **CY8CKIT-062-BLE** for the **CY8CKIT-062-BLE PSoC™ 6 BLE Pioneer Kit**. Click **Next**.

In the next dialog, **Starter Application**, select **EmptyPSoC6App**, and change the **Name**. CE213903 uses "PSoC6DfuApp0Uart", "PSoC6DfuApp0I2c", and "PSoC6DfuApp0Spi"; choose a name that indicates that this is the application that does the DFU. You can also indicate the host communication channel. Click **Next**. In the Summary dialog, confirm your selections and click **Finish**.

2. Configure App0 as a DFU

To add DFU capability to an application, you must:

- Add a host communication channel to your design
- Add other peripherals as needed
- Add DFU code
- **Add a communication channel** Open design. `modus` in project `<name>_config`. Select a channel type, and configure it to work with the CY8CKIT-062-BLE kit:
 - **For UART:** Add a UART to SCB 5, with the following changes from the default: Oversample = 12, Clock = any convenient clock, RX = P5[0], TX = P5[1]. You can optionally assign names to the UART and the UART pins
 - **For I²C:** Add an I²C to SCB 3, with the following changes from the default: Data Rate (kbps) = 400, Clock = any convenient clock, SCL = P6[0], SDA = P6[1]. You can optionally assign names to the I²C and the I²C pins
 - **For SPI:** Add a SPI to SCB 6, with the following changes from the default: Clock = any convenient clock, SCLK = P12[2], MOSI = P12[0], MISO = P12[1], SS1 = P12[4]. You can optionally assign names to the SPI and the SPI pins
- **Add other peripherals as needed** For compatibility with CE213903, configure pins for kit LED and button support:
 - Configure pin P0[3] as follows: name = KIT_RGB_R, Drive Mode = Strong Drive Input Buffer off. Confirm that the Initial Drive State = High (1)
 - Configure pin P0[4] as follows: name = KIT_BTN1, Drive Mode = Resistive Pull-Up Input buffer on. Confirm that the Initial Drive State = High (1)

5 PSoC™ 6 application notes

- DRAFT**
- **Add DFU code** Right-click project <name>_mainapp and select **ModusToolbox™ Middleware Selector**. Select Middleware **DFU SDK Core**, and **DFU SDK transport UART** or one of the other transport options
Do the same for <name>_mainapp_cm0p and select only **DFU SDK core**. Required source .c, .h, and linker script (e.g., .ld) files are automatically added to both projects
For I²C and SPI channels, edit the file dfu_user.c, in the <name>_mainapp project Source folder, as follows:
 - Change #include "transport_uart.h" to:
#include "transport_i2c.h" or
#include "transport_spi.h".
 - Change five instances of "UART_Uart" to "I2C_I2c" or "SPI_Spi".
 - Integrate DFU code into the main.c files in <name>_mainapp and <name>_mainapp_cm0p. The easiest way to do this is to copy and paste the code from the code example [CE213903](#), and then modify the code for your application
Right-click project <name>_mainapp and select **Properties**. In C/C++ >**Build > Settings > Tool Settings > > GNU ARM Cross C Linker > General**, change the file in the **Script files** entry to dfu_cm4.1d, to link using the custom script in the project. Do the same for project <name>_mainapp_cm0p, for dfu_cm0p.1d.
 - **Build the project** The easiest way to do this is to select <name>_mainapp, then in the Quick Panel click **Build <name> Application**

3. Add App1

As noted previously, with ModusToolbox™ IDE you can add applications to the same workspace as the DFU application, or they can be in separate workspaces, folders, or both. These instructions show how to add App1 to the same workspace as the DFU App0

Note: You can create any number of installable applications that work with the same DFU system. Before getting started with developing applications, developing a plan for DFU and applications for your overall system development needs is recommended. See [Determine the applications in your system](#).

Create a new application in the same manner as described previously. Select the same kit as was done for App0. Change the name; CE213903 uses "PSoC6DfuApp1". Choose a name that indicates that this is the downloadable application

4. Configure App1 as an installable application

- **Add other peripherals as needed** Edit design. modus for this application. For compatibility with CE213903, configure pins for kit LED and button support:
Configure pin P0[3] as follows: name = KIT_RGB_R, Drive Mode = Strong Drive Input Buffer off.
Confirm that the Initial Drive State = High (1)
Configure pin P0[4] as follows: name = KIT_BTN1, Drive Mode = Resistive Pull-Up Input buffer on.
Confirm that the Initial Drive State = High (1)
- **Add DFU code** Right-click project <name>_mainapp and select **ModusToolbox™ Middleware Selector**. Select Middleware **DFU SDK Core**. Do the same for <name>_mainapp_cm0p. Required source .c, .h, and linker script (e.g., .ld) files are automatically added to both projects. Adding DFU SDK Core to the application is done only to enable transfer of control to App0

5 PSoC™ 6 application notes

DRAFT

Edit the files dfu_cm4.1d in project <name>_mainapp and dfu_cm0p.1d in project <name>_mainapp_cm0p as follows. This changes the application # for App1, and properly locates flash and RAM memory for App1:

```

/*
 * DFU SDK specific: aliases regions, so the rest of code does not use
 * application specific memory region names
 */
REGION_ALIAS("flash_core0", flash_app1_core0);
REGION_ALIAS("flash", flash_app1_core1);
REGION_ALIAS("ram", ram_app1_core1);

/* DFU SDK specific: sets an app Id */
__cy_app_id = 1;

```

Note: *It is important to not change anything else in these. Id files. With the exception of the above edits, they must be the same as the App0. Id files.*

Integrate DFU code into the main.c files in <name>_mainapp and <name>_mainapp_cm0p. The easiest way to do this is to copy and paste the code from the code example CE213903, and then modify the code for your application

Copy the file dfu_postbuild.bash from the code example into the project <name>_mainapp. See [Appendix F](#).

Right-click project <name>_mainapp_cm0p and select **Properties**. In **C/C++ Build > Settings > Tool Settings > GNU ARM Cross C Linker > General**, change the file in the **Script files** entry to dfu_cm0p.1d, to link using the custom script in the project

Do the same for project <name>_mainapp, for dfu_cm4.1d. Then, in the same window select **Build Steps**, and replace the **Post-build steps Command** with the following:

```

${cy_sdk_install_dir}/tools/modus-shell-1.0/bin/bash --norc --noprofile
${workspace_loc:/PSoC6DfuApp1_mainapp}/dfu_postbuild.bash
${cy_sdk_install_dir}/tools/cymcuelftool-1.0/bin/cymcuelftool
${workspace_loc:/PSoC6DfuApp1_mainapp_cm0p}/${config_name}:
PSoC6DfuApp1_mainapp_cm0p/PSoC6DfuApp1_mainapp_cm0p.elf
${workspace_loc:/PSoC6DfuApp1_mainapp}/${config_name}:PSoC6DfuApp1_mainapp/
PSoC6DfuApp1_mainapp.elf ARM_CM4

```

to create the output. cyacd2 file, which is used to download the application for DFU. See [Appendix C](#).

- **Build the project.** The easiest way to do this is to select <name>_mainapp, then in the Quick Panel click **Build <name> Application**.

5. Test the DFU and application

Program App0 into a [CY8CKIT-062-BLE PSoC™ 6 BLE Pioneer Kit](#). Then use the Device Firmware Update Host Tool (DHT) to download App1 to the kit; see tool usage instructions in [Appendix A](#).

You can test the DFU and application using the instructions in the [CE213903](#) document, Operation section

5.16.5.2.2 PSoC Creator instructions

1. Create App0 and the workspace.

5 PSoC™ 6 application notes

You can create a PSoC™ Creator project and its containing workspace at the same time. In PSoC™ Creator, select **File > New > Project....** The Create Project window with the Select project type panel appears, as [Figure 578](#) shows.

- DRAFT**
- a. Click **Target device**
- b. In the first drop-down menu, select **PSoC™ 6**
- c. In the next drop-down menu, select **CY8C6347BZI-BLD53**. This is the PSoC™ 6 MCU device that is installed in the [CY8CKIT-062-BLE PSoC™ 6 BLE Pioneer Kit](#)
- d. Click **Next**

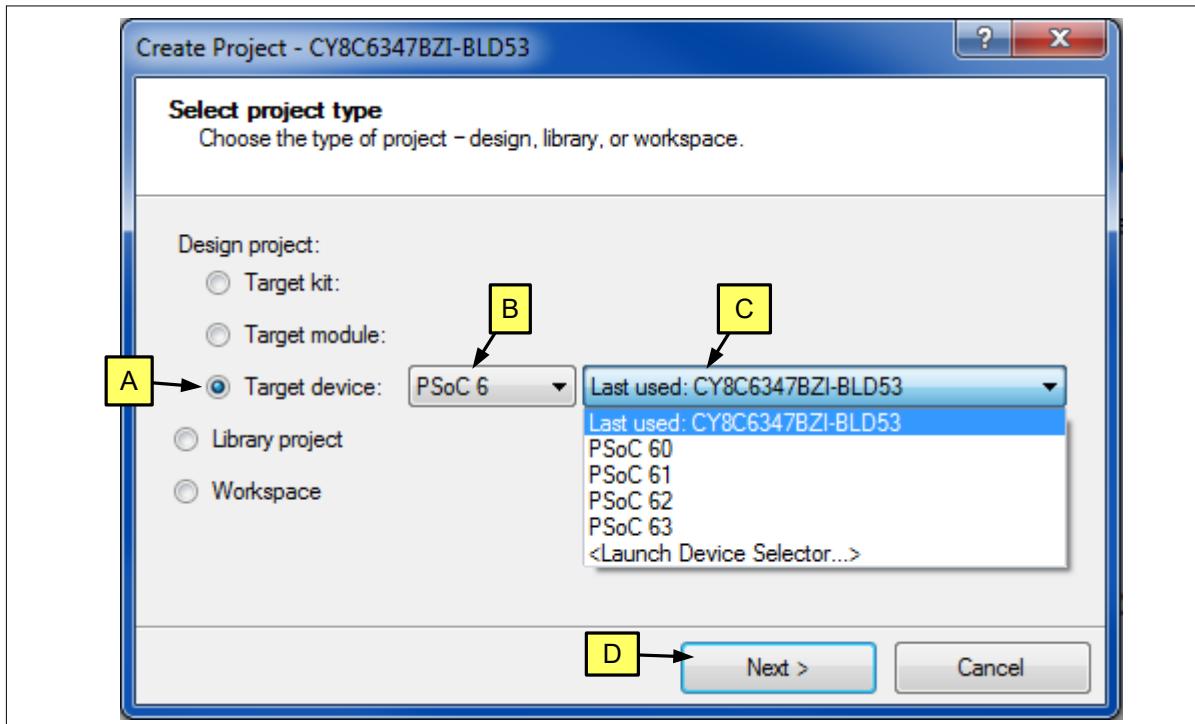


Figure 578 PSoC™ Creator select the target device

Click **Next** on the next two panels, **select project template** and **Set target IDE(s)**. The Create Project panel appears, as [Figure 579](#) shows.

Enter the **Workspace name** and select its **Location**. Set **Project name** to “app0”. Click **Finish**.

A new folder is created in the indicated location; the folder name is the same as the workspace name. A folder `app0.cydsn` is created within the workspace folder; the `.cydsn` folder contains all project files.

5 PSoC™ 6 application notes

DRAFT

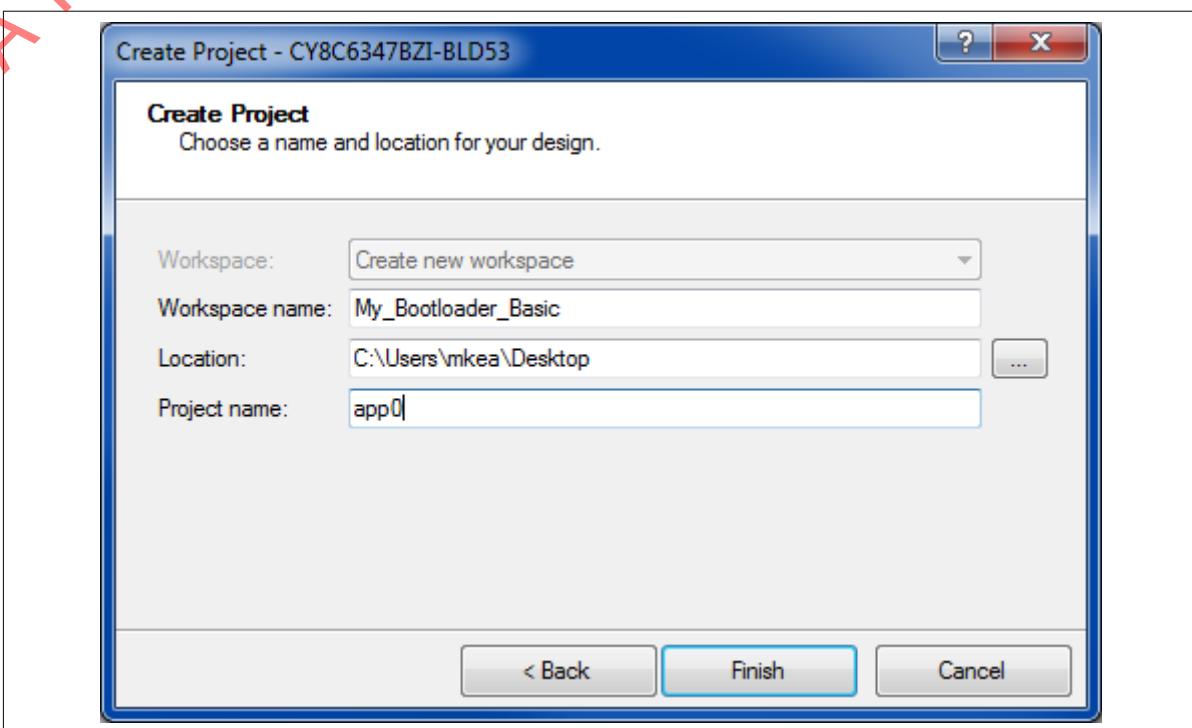


Figure 579 Create the PSoC™ Creator project

The new workspace and project files are shown in the Workspace Explorer window; see [Figure 574](#).

2. Configure App0 as a Bootloader

To add bootloader capability to an application, you must:

- Add a communication Component to the project schematic. Use this Component to communicate with your bootloader host
 - Add other Components to the project schematic as needed for your application
 - Add Bootloader SDK and template files to the project
- a. **Add a communication Component.** Open or double-click the project schematic (file TopDesign.cysch in the Workspace Explorer window)
Open the Component Catalog, and navigate to your desired communication Component, for example UART, as [Figure 580](#) shows. Drag the Component onto the schematic

5 PSoC™ 6 application notes

DRAFT

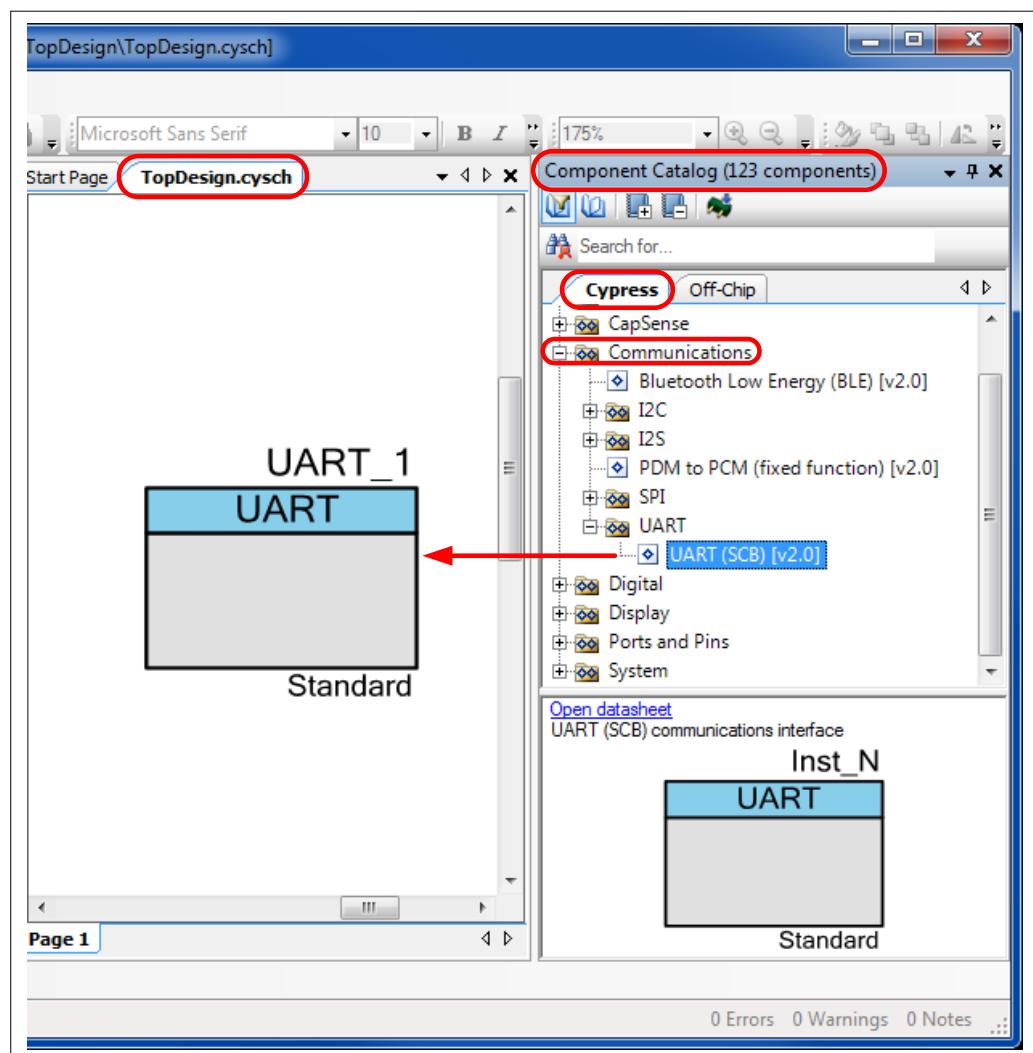


Figure 580 Add a PSoC™ Creator communication component to a bootloader project

Double-click the Component on the schematic to configure its parameters, such as baud rate, number of bits, etc., as [Figure 581](#) shows.

[Figure 581](#) shows the changed parameter settings for this code example; note that **Name** is changed from UART_1 to UART, I2C_1 to I2C, or SPI_1 to SPI. This is recommended to work with the default Bootloader SDK files that are copied to your project later.

When done configuring the Component, click **OK**.

5 PSoC™ 6 application notes

DRAFT

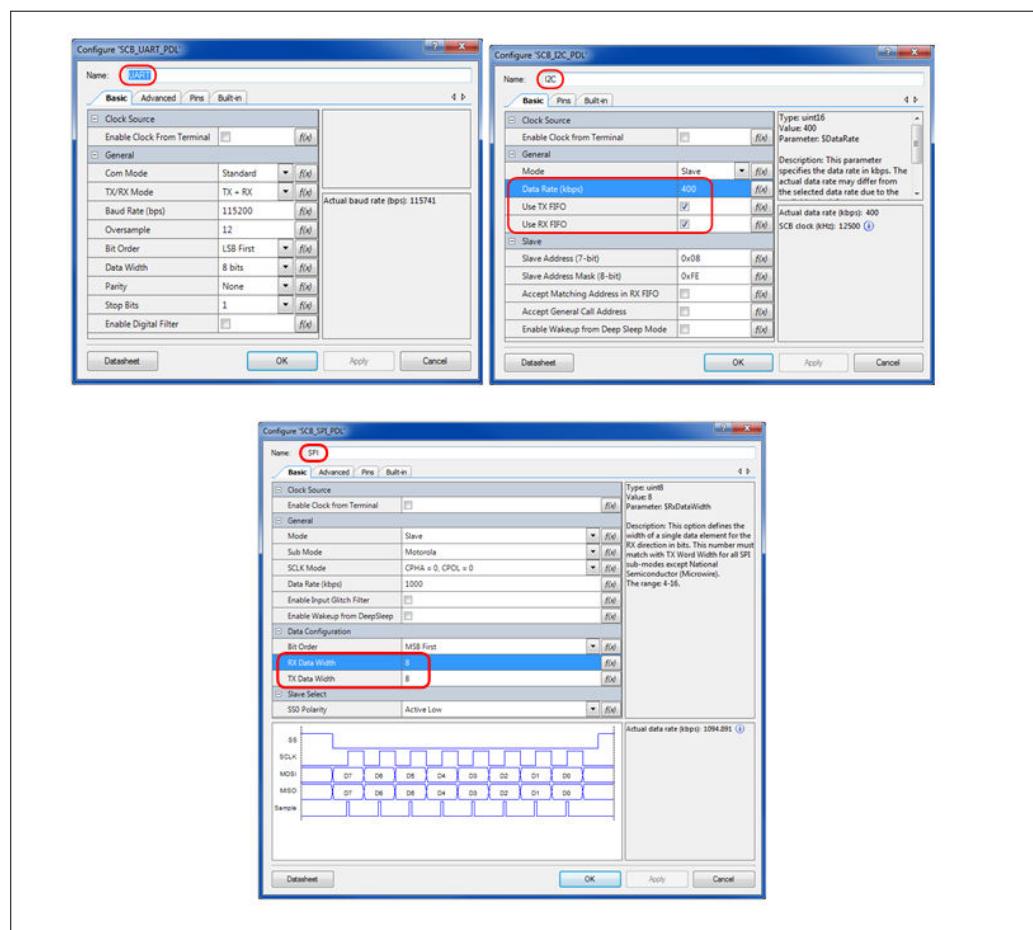


Figure 581 Configure the PSoC™ Creator communication component: UART, I2C, or SPI

- b. Add other Components** Add other PSoC™ Creator Components such as LED and button pins to the project schematic. The easiest way to do this is to copy and paste portions of the project schematic from the code example [CE213903](#), and then modify the schematic as needed for your application

In the Design Wide Resources window, **Pins** tab, connect the UART, I2C, SPI, and other Component pins to the appropriate physical pins. Use [CE213903](#) and the guide for your kit for instruction

- c. Add Bootloader code to the project** First, incorporate the Bootloader SDK into the project – see [Figure 573](#)

Then select **Build > Generate Application**. When done, the project should look like [Figure 574](#), with the addition of some startup and other non-bootloader files

For I²C and SPI bootloaders, edit the file `bootload_user.c`, in the project Shared Files folder, as follows:

- Change `#include "transport_uart.h"` to:
`#include "transport_i2c.h"` or
`#include "transport_spi.h".`
- Change five instances of “UART_Uart” to “I2C_I2c” or “SPI_Spi”.

Add bootloader code to the `main_cm0p.c` and `main_cm4.c` files. The easiest way to do this is to copy and paste the code from the code example [CE213903](#), and then modify the code for your application

- d. Build the project** Change the project build settings to use the template linker script files; see [Figure 575](#). Then select **Build > Build app**

5 PSoC™ 6 application notes

~~DRAFT~~ 3. Add App1

As noted previously, with PSoC™ Creator, you can add other projects – applications – to the same workspace as the bootloader application, or they can be in separate workspaces, folders, or both. These instructions show how to add App1 to the same workspace as the bootloader App0

Note: You can create any number of installable applications that work with the same bootloader. Before getting started with developing applications, developing a plan for bootloader and applications for your overall system development needs is recommended. See [Determine the applications in your system](#).

In the PSoC™ Creator Workspace Explorer window, right-click the **Workspace**, then select **Add > New Project...**, as [Figure 582](#) shows:

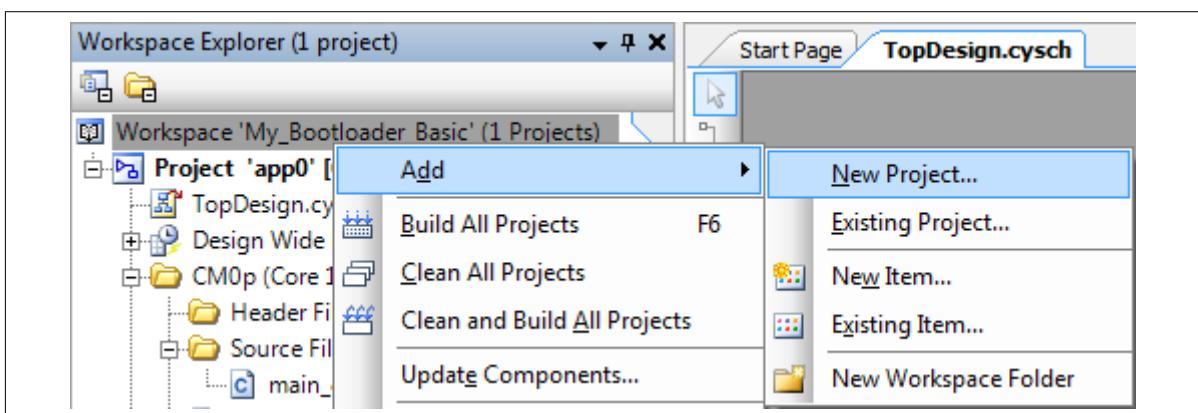


Figure 582 Add a new project to a PSoC™ Creator workspace

The Create Project window with the Select project type panel appears; see [Figure 578](#). Make sure that the device selected is the same as for App0. Click **Next**

Click **Next** on the next two panels: **Select project template** and **Set target IDE(s)**. The Create Project panel appears

Set **Project name** to “app1”, as [Figure 583](#) shows. The default Location is the workspace folder. Click **Finish**

A new app1.cydsn folder is created within the workspace folder; the .cydsn folder contains all project files

5 PSoC™ 6 application notes

DRAFT

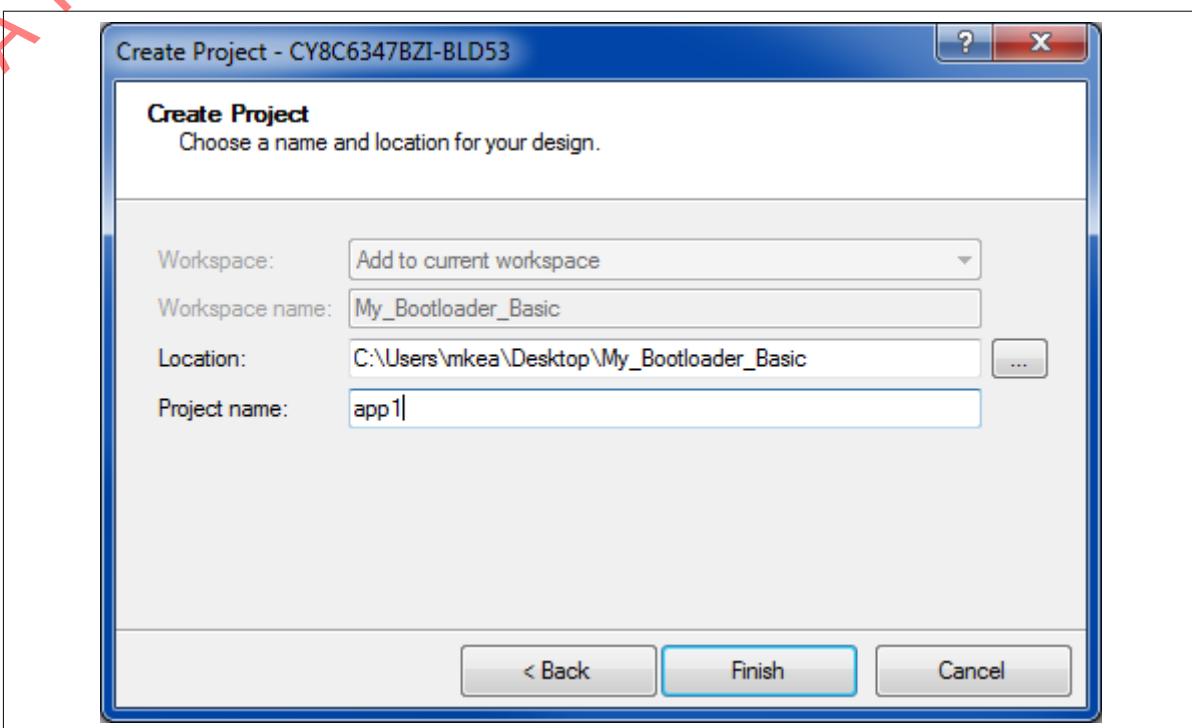


Figure 583 Create a second PSoC™ Creator project

4. Configure App1 as an installable application

Note: App1 should be built with the same toolchain (GCC or MDK) as App0, or application transfer may fail

- Add other Components such as LED and button pins to the project schematic** The easiest way to do this is to copy and paste portions of the project schematic from the code example [CE213903](#), and then modify the schematic for your application
In the Design Wide Resources window, **Pins** tab, connect the Component pins to the appropriate physical pins. Use [CE213903](#) and the guide for your kit for instruction
- Add Bootloader SDK and template files to the project** Incorporate the Bootloader SDK into the project; see [Figure 573](#). Select only the **Core** box, because App1 is a downloadable application and does not have bootloader capabilities
Then, select **Build > Generate Application**. When done, the project should look like [Figure 574](#), with the addition of some startup and other non-bootloader files. Change the project build settings to use the template linker script files; see [Figure 571](#)
Add a post-build batch file to the Shared Files folder, as described in [Section 4.3.2 Step 8](#). Then add a post-build command to the Cortex-M4 build – see [Figure 576](#)
- Edit files** Edit the linker script files as described in [Section 4.3.2 Step 6](#).
Add bootloader code to the `main_cm0p.c` and `main_cm4.c` files. The easiest way to do this is to copy and paste the code from the code example [CE213903](#), and then modify the code for your application
Make sure that an array has been added to the `main_cm4.c` file for the application signature, as described in [Section 4.3.2 Step 8](#)
- Build the project.** After all build setting changes and file edits are complete, select **Build > Build app1**

5. Test the bootloader and applications

Program app0 into a [CY8CKIT-062-BLEPSoC™ 6 BLE Pioneer Kit](#). Then use the Bootloader Host Program (BHP) to bootload app1 to the kit; see BHP usage instructions in [Appendix A](#)

~~5 PSoC™ 6 application notes~~

You can test the bootloader and applications using the instructions in the [CE213903](#) document, Operation section

5.16.5.3 PSoC™ 6 MCU BLE bootloaders

This section shows how to build three different BLE bootloaders in code examples:

- [CE216767](#), PSoC™ 6 MCU with Bluetooth Low Energy (BLE) Connectivity Bootloader
- [CE220959](#), PSoC™ 6 MCU BLE Bootloader with External Memory
- [CE220960](#), PSoC™ 6 MCU BLE Bootloader with Upgradeable Stack (see [this subsection](#))

These code examples are similar to the [PSoC™ 6 MCU basic DFUs](#), with some additional features:

- CE216767 downloads a new application directly into flash, as [Figure 584](#) shows

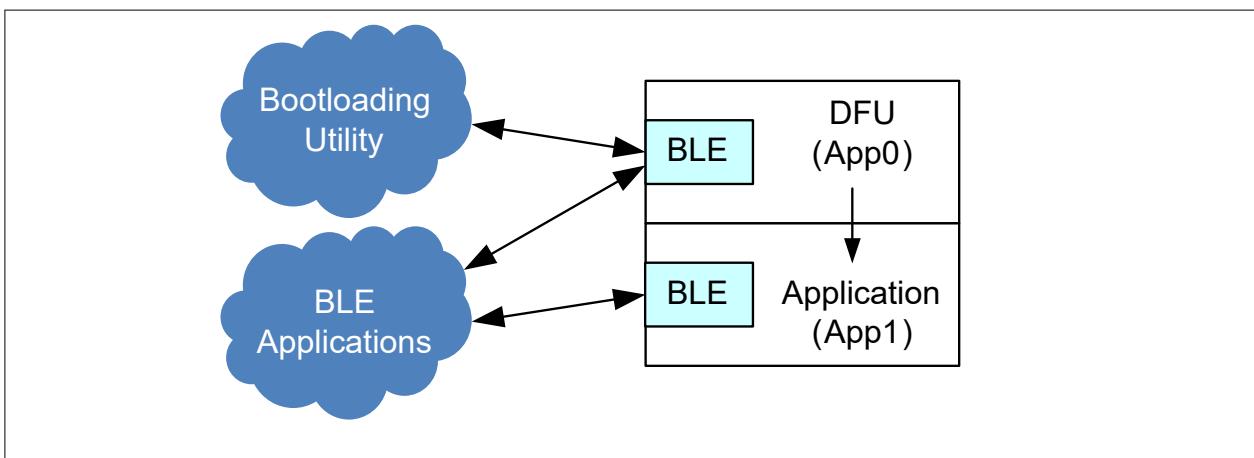


Figure 584 Data and control flow for CE216767

- CE220959 downloads a new application into temporary storage in an external NVM (kit IC U4, Cypress 512-Mbit serial NOR flash), and then copies it to its final destination in the PSoC™ 6 MCU device flash, as [Figure 585](#) shows

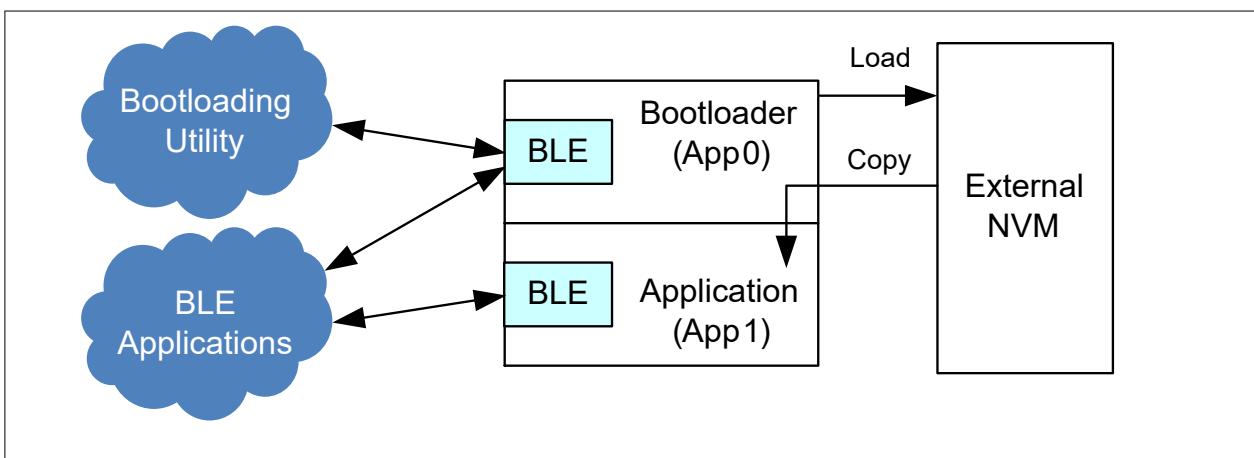


Figure 585 Data and control flow for CE220959

- The bootloader (App0) and the application (App1) each have their own separate copy of the BLE stack code, as [Figure 584](#) and [Figure 585](#) show. Each copy occupies more than 128 KB of flash
For a bootloader with a single shared BLE stack, see [BLE bootloader with upgradeable stack](#).
- The BLE stack code can be executed by either CPU or by both CPUs; this is a selectable option in the BLE Component configuration dialog. The code examples demonstrate this feature; in App0 the CM4 CPU executes the BLE stack, in App1 the CM0+ CPU executes the BLE stack

~~5 PSoC™ 6 application notes~~

- App0 and App1 enable different services in their BLE Component configurations:
 - App0: Bootloader and Immediate Alert Service (IAS). The IAS service is implemented when the application is not bootloading
 - App1: Human Interface Device (HID) and Immediate Alert Service (IAS)
- Both applications use the IAS to transfer control from one application to the other
- The code examples make extensive use of [CY8CKIT-062-BLE PSoC™ 6 BLE Pioneer Kit](#) resources. All three RGB LEDs, the user button SW2, the BLE subsystem, and the USB-UART bridge are all used by both applications (App0 and App1)
- To build the code examples, do the following. For more information on these steps, refer to [PSoC™ 6 MCU basic DFUs](#) or [How to use the SDK](#).

1. Create App0 and the workspace, as [Figure 586](#) shows

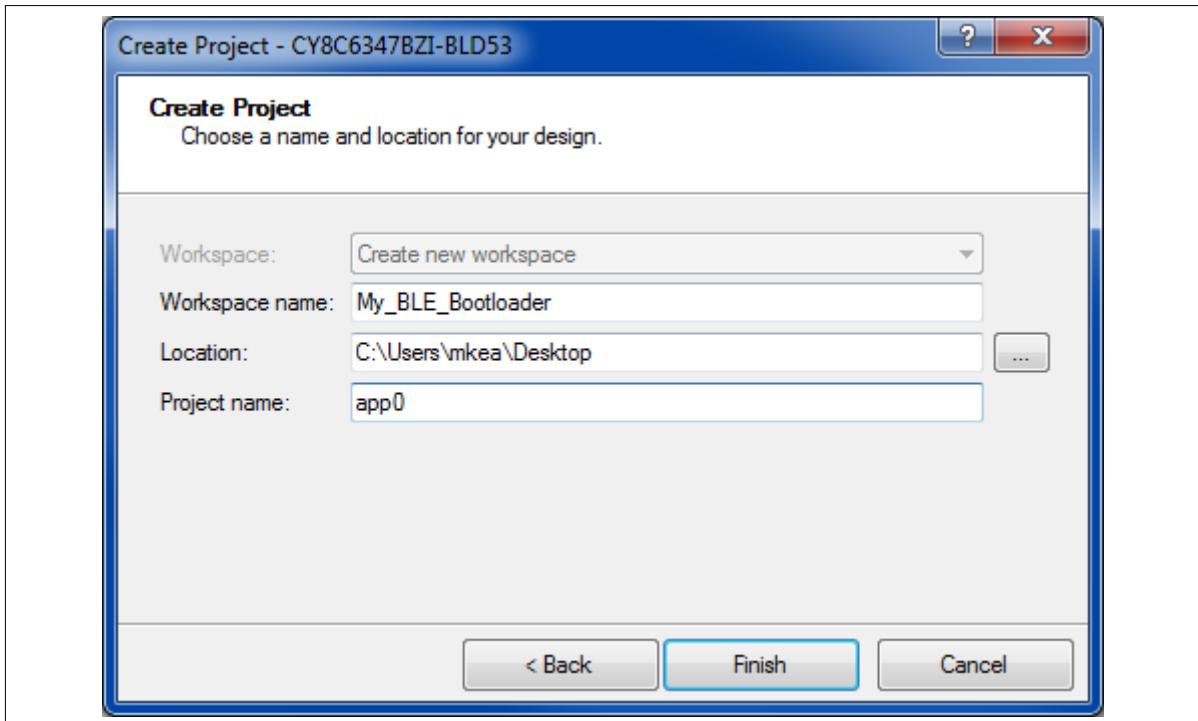


Figure 586 PSoC™ Creator create BLE bootloader App0

2. Add a BLE Component to the App0 Top Design schematic and configure the Component according to the BLE Component Configuration section in either code example. Specifically, make the following changes from the default configuration:
 - (optional, but recommended to work with default Bootloader SDK files) Change the Component name to 'BLE'
 - General tab** (see [Figure 587](#)): CPU core: **Single core (Complete Component on CM4)**

5 PSoC™ 6 application notes

DRAFT

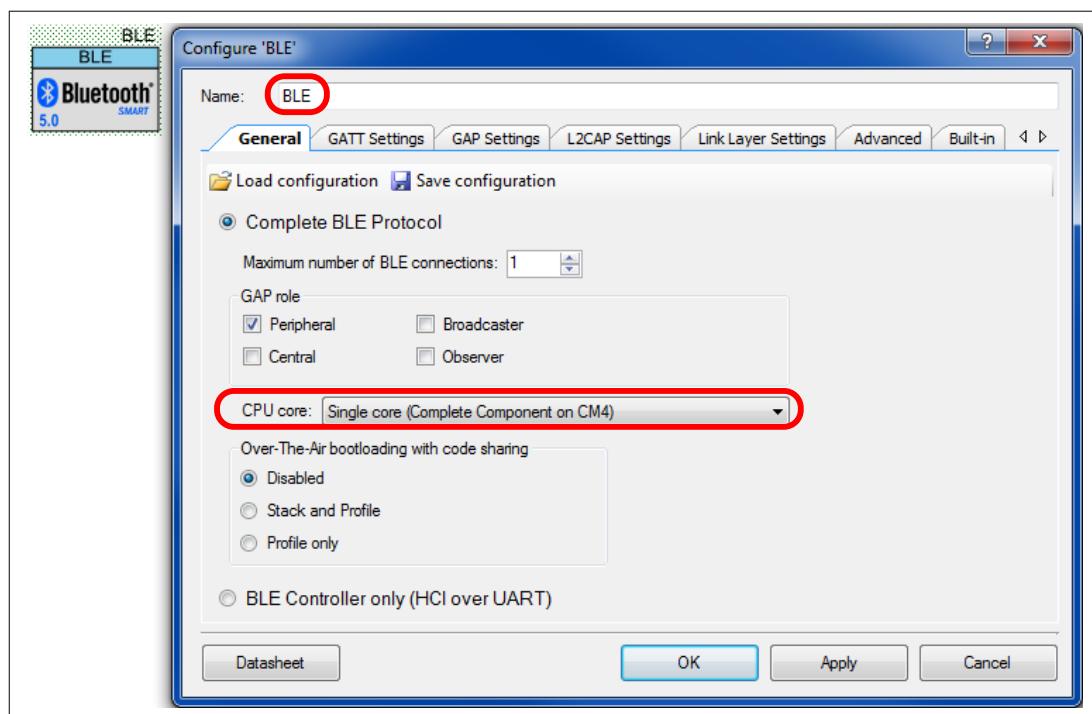


Figure 587 PSoC™ Creator BLE component, General Tab configuration

- GATT Settings tab (see [Figure 588](#)):
 - Generic Access, Peripheral Preferred Connection Parameters:
 - a. Minimum Connection Interval: **0x000C**
 - b. Maximum Connection Interval: **0x000C**
 - c. Connection Supervision Timeout Multiplier: **0x00C8**

These intervals are selected to minimize the bootloading time. Adjust as needed for your application.
 - Right-click the Server node in the GATT tree, and add the Bootloader service. Optionally, add the Immediate Alert service; the code examples use this service to transfer control between applications. There is no need to edit either service's characteristics
 - Attribute MTU size (bytes): 512

5 PSoC™ 6 application notes

DRAFT

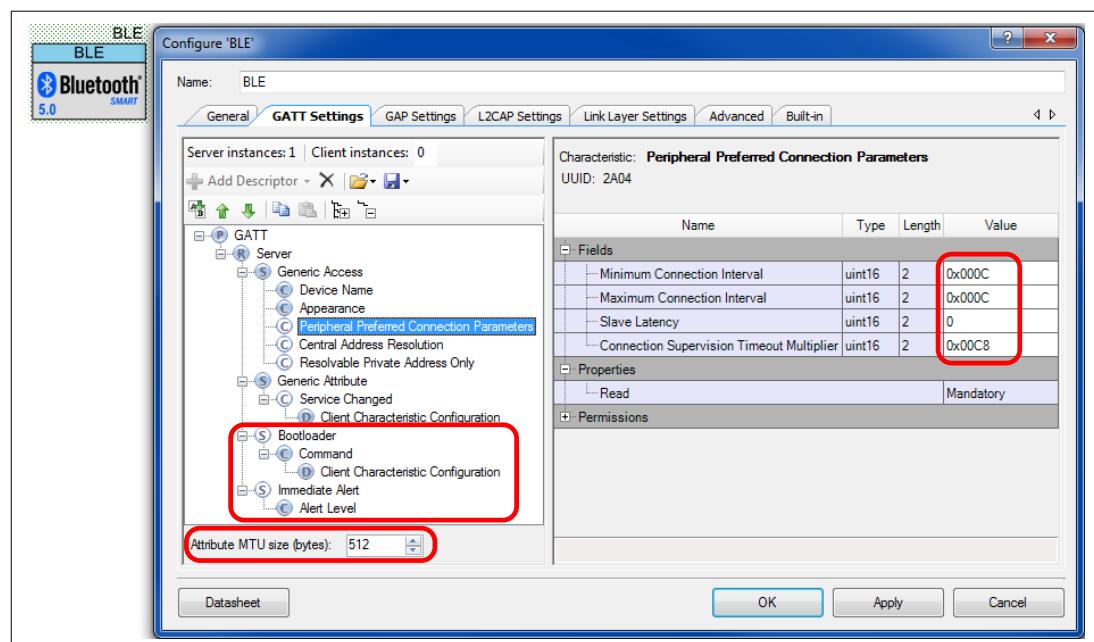


Figure 588 PSoC™ Creator BLE component, GATT Settings tab configuration

- GAP Settings tab:
 - General, Device Name: The code examples use “BLE Bootloader”; use different text as needed for your application.
 - Peripheral Configuration 0, Advertisement packet: **Local Name** checked and set to **Complete**.
 - Security configuration 0 (see Figure 589), Security level: **Unauthenticated pairing with encryption**.
 - Security configuration 0, I/O capabilities: **No Input No Output**.
 - Security configuration 0, Bonding requirement: **No Bonding**.

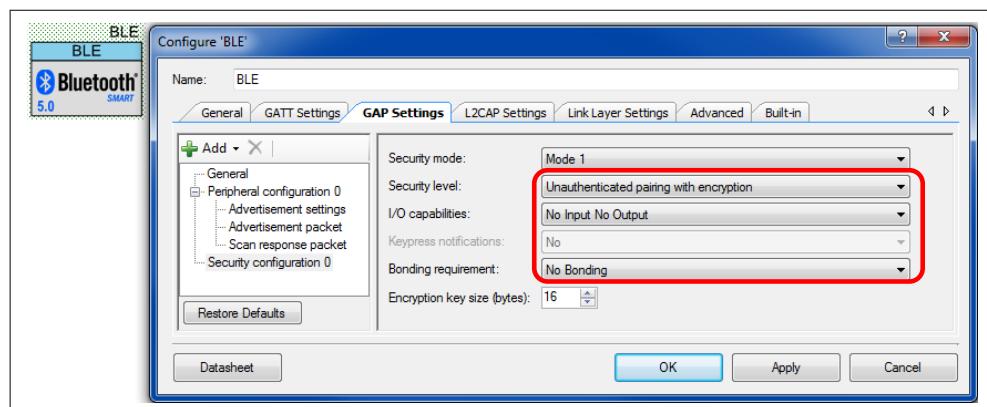


Figure 589 PSoC™ Creator BLE component, GAP Settings tab configuration

- Link Layer Settings tab:
 - Link layer max TX and RX payload size (bytes): 251, for faster bootloading
- 3. For [CE220959](#), add an SMIF Component to the App0 schematic, and configure the Component according to the SMIF Component Configuration section. Specifically, make the following changes from the default configuration:
 - (optional, but recommended to work with CE220959 files) Name: SMIF

5 PSoC™ 6 application notes

- DRAFT**
- Select **SMIF Datalines [2:3]**. The external memory IC has a quad-SPI (four data-line) interface with the PSoC™ 6 MCU device. The **SMIF Datalines [0:1]** box is selected as a default
 - Unselect **Generate code from cy_smif.cysmif file**. Files `cy_smif_memconfig.h/c` and `smif_mem.h/c` are already provided with the code example
4. As needed, copy other Components – i.e., LEDs and button – from either the [CE216767](#) or [CE220959](#) top design schematic to your top design schematic
5. In the Design Wide Resources window:
- **Pins** tab: Connect the Component pins to the appropriate physical pins. Note that the BLE Component does not use any pins
 - **Clocks** tab: Enable the WCO, and source Clk_LF and Clk_Bak from WCO. This required for operating the BLE Component in certain modes; see the BLE Component datasheet for details
 - For [CE220959](#), enable Clk_HF2 and set its frequency to 50 MHz. The easiest way to do this is to select the divide by 2 option in the Clk_HF2 section. The frequency limit is required for correct operation of the external memory interface
6. Configure App0 as a bootloader in the project **Build Settings** – see [Figure 573](#). Select **BLE** for the communication channel.
7. Select **Build > Generate Application**. This creates the bootloader linker files, e.g., `bootload_cm4.1d`.
8. Go back to the project **Build Settings** and set the **Custom Linker Script** for CM0+ and CM4 to the respective bootloader linker script files – see [Figure 575](#).
9. Edit the flash and RAM memory region sizes in `bootload_common.1d`, as follows:

```
MEMORY
{
    flash_app0_core0  (rx)  : ORIGIN = 0x10000000, LENGTH = 0x10000
    flash_app0_core1  (rx)  : ORIGIN = 0x10010000, LENGTH = 0x30000
    flash_app1_core0  (rx)  : ORIGIN = 0x10040000, LENGTH = 0x32000
    flash_app1_core1  (rx)  : ORIGIN = 0x10072000, LENGTH = 0x02000
    .
    .
    ram_app0_core0    (rwx) : ORIGIN = 0x08000100, LENGTH = 0x1F00
    ram_app0_core1    (rwx) : ORIGIN = 0x08002000, LENGTH = 0x8000
    .
    .
    ram_app1_core0    (rwx) : ORIGIN = 0x08000100, LENGTH = 0x1FF00
    ram_app1_core1    (rwx) : ORIGIN = 0x08020000, LENGTH = 0x20000
    .
}
```

The default `bootload_common.1d` allocates an equal amount of flash and RAM to each CPU in each application. The allocations must be adjusted for these code examples because:

- The BLE stack size is large
- App1 size should be minimized to reduce bootloading time

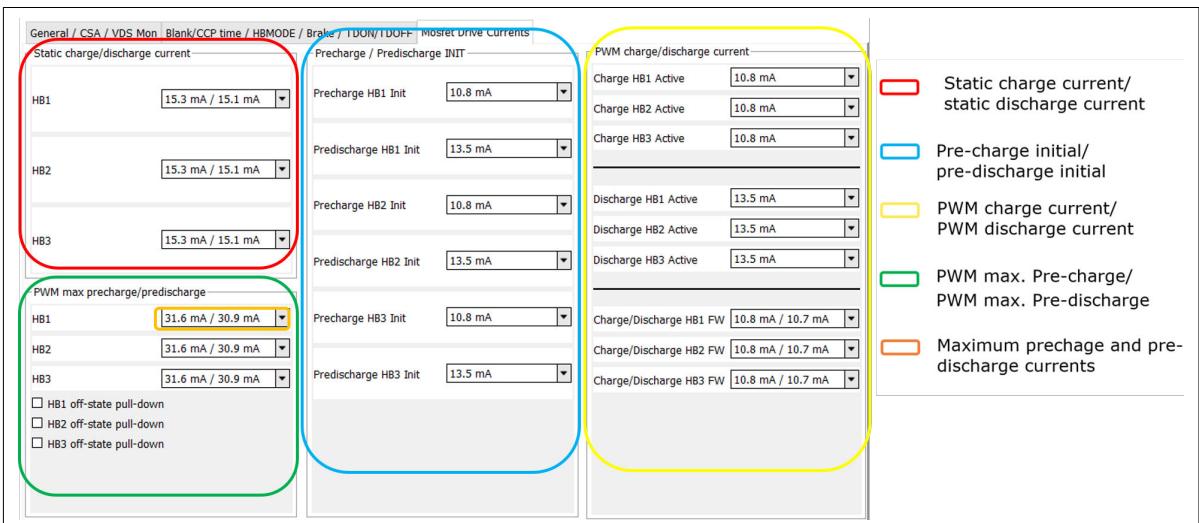
Note that the same changes are done for App1 in the code examples. Similar memory layouts exist for the MDK and IAR linker scripts.

10. Copy and paste files `debug.h/c`, `ias.h/c`, and `bootload_user.h/c` from your [CE216767](#) or [CE220959](#) folder to your `app0.cydsn` folder. Overwrite the existing default `bootload_user.h/c` files. In addition, for [CE220959](#), copy and paste `cy_smif_memconfig.h/c` and `smif_mem.h/c` files
- The `debug` and `ias` (Immediate Alert Service) files are unique to the code examples and may be optional for your application

5 PSoC™ 6 application notes

- The bootload_user files have already been edited to call the BLE transport functions instead of the default UART transport functions
- For CE220959, the SMIF files define the external memory configuration and provide functions to access external memory

11. In the PSoC™ Creator Workspace Explorer window, include or move the files from the previous step to the project CM4 Header Files and Source Files folders, as Figure 590 shows

**Figure 590****Source files included in PSoC™ Creator CM4 folders**

If these files are in the Shared Files folder, they may be included in the CM0+ build and compile errors may result

12. Copy CM0+ and CM4 main code from **CE216767** or **CE220959** to your main files. Update the main and other source files as needed for your application

13. Build app0, and program the kit with app0

You can test your app0 by installing app1 from CE216767 or CE220959. Follow the instructions in the code example document, Operation section

14. If you build your own app1, remember to edit the following files:

- Three linker script files as described in 9. and in [Section 4.3.2 Step 6](#).
- The batch file as described in [Section 4.3.2 Step 8](#).

5.16.5.3.1 BLE bootloader with upgradeable stack

This code example is different from [the other BLE bootloader code examples](#) in that it has three applications, as Figure 591 shows.

5 PSoC™ 6 application notes

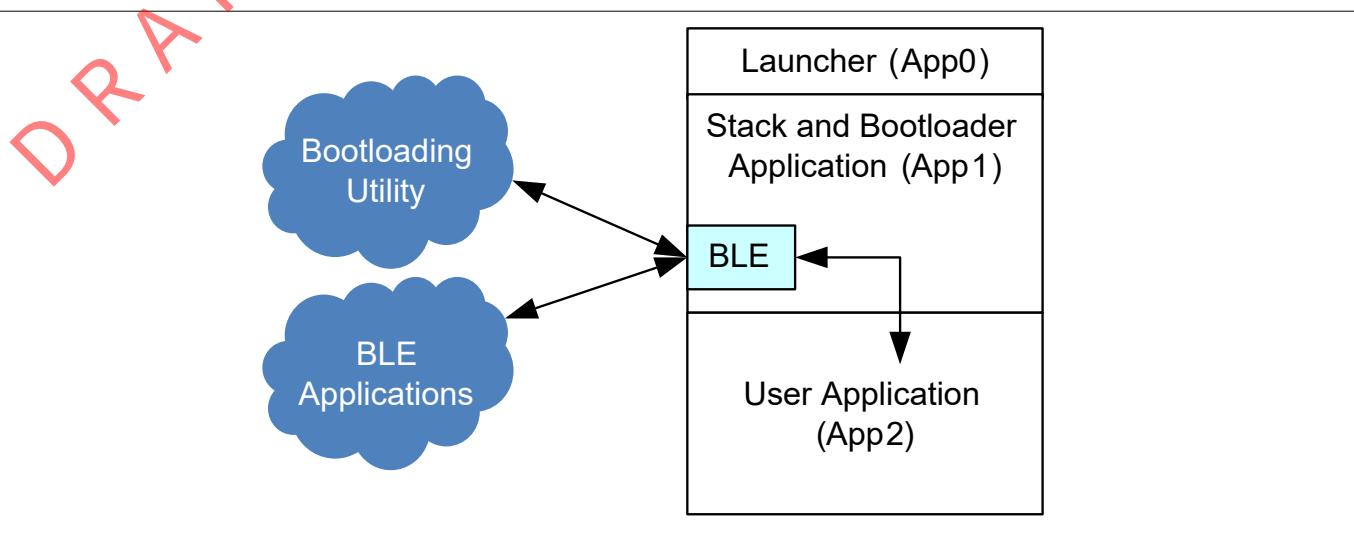


Figure 591 Data and control flow for CE220960

In this example App0 does not do bootloading; it is solely a launcher application. It also copies a stack update from a temporary location to the proper location and starts either the stack or the user application. App0 cannot be updated by OTA bootloading.

App1 is the stack application. It contains the bootloader and the BLE stack. The bootloader can download an update to the user application or to the stack. Updating the stack requires placing the update in a temporary location and switching to the launcher for copying, because the stack cannot overwrite itself.

App2 is the user application. It has a PSoC™ Creator BLE Component, but it contains only profiles without the supporting stack code. Required stack code and variables are shared from the stack application, considerably reducing the size of the user application.

The steps to build this code example are similar to those for the other BLE code examples; the main differences are that you create three projects, and the BLE configurations are different:

- Create App0 and the workspace, as [Figure 586](#) shows
- As needed, copy Components – i.e., LEDs and button – from the [CE220960](#) App0 top design schematic to your top design schematic
- In the Design Wide Resources window, **Pins** tab, connect the Component pins to the appropriate physical pins. Use [CE220960](#) and the guide for your kit for instructions
- Configure App0 as a bootloader in the project **Build Settings** – see [Figure 573](#). You must do this to enable the copy application and transfer control functions – select only the **Core** and **App type** boxes. Do not select any of the communication channel boxes
- Select Build > Generate Application. This creates a number of files, including the bootloader linker files, e.g., `bootload_cm4.1d`
- Go back to the project **Build Settings** and set the **Custom Linker Script** for CM0+ and CM4 to the respective bootloader linker script files; see [Figure 575](#). Also, add to the **CM4 User Commands** a call to the `post_build_core1.bat` file; see [Figure 576](#)
- Update the files, and copy the new files, listed in [Table 141](#) from the corresponding files in [CE220960](#). Add code as needed for your application

5 PSoC™ 6 application notes

~~DRAFT~~
Table 141

CE220960PSoC™ Creator App0 files modified from the bootloader SDK default

File	Description of modification from the default
bootload_common.ld, bootload_cm0p.ld, bootload_cm4.ld	Modified memory allocations for the three applications
bootload_mdk_common.h, bootload_mdk_symbols.c	
bootload_cm0p.scat, bootload_cm4.scat	
bootload_user.h, bootload_user.c	Modified number of applications in metadata
main_cm0p.c, main_cm4.c	Addition of code to do the code example functions

New files; copy from the code example, and add to the Shared Files folder

- | | |
|----------------------|---|
| post_build_core1.bat | Copies output to a build location for App1 and App2 |
|----------------------|---|
- Add a new project App1 to the workspace. Add Components and files to the project, and configure it, in the same manner as App0 in the [other BLE code examples](#); except in the BLE configuration dialog, select **Stack and Profile** for **Over-The-Air bootloading with code sharing**
 - Use CE220960 App1 for guidance. Note that in CE220960 App1 has two post-build batch files, one for each core
 - Test using the instructions in CE220960
 - Add a new project App2 to the workspace. Add Components and files to App2, and configure it, in the same manner as App0 in the [other BLE code examples](#), except in the BLE configuration dialog, select **Profile Only** for **Over-The-Air bootloading with code sharing**. Use CE220960 App2 for guidance

5.16.5.4 PSoC™ 6 MCU dual-application bootloader

This section shows how to build the BLE bootloader in the code example [CE221984](#). This code example is like the [basic bootloaders code example](#), except that there are two downloadable applications instead of one. There are three applications; App0 is the bootloader and App1 and App2 are the downloadable applications.

The bootloader transfers control to one of the applications in either a basic mode or a factory default (“golden image”) mode. In factory default mode, the bootloader (App0) does not overwrite an installed and valid App1; an attempt to do so results in an error message. In basic mode, either application can be overwritten.

In this code example, the bootloader uses an I²C communication channel. Changing to a different communication channel is simple; for more information, refer to one of the [previous](#) sections.

To build this code example, do the following steps. For more information on these steps, refer to [PSoC™ 6 MCU Basic Bootloaders](#) or [How to use the SDK](#).

1. Create App0 and the workspace, as [Figure 592](#) shows

5 PSoC™ 6 application notes

DRAFT

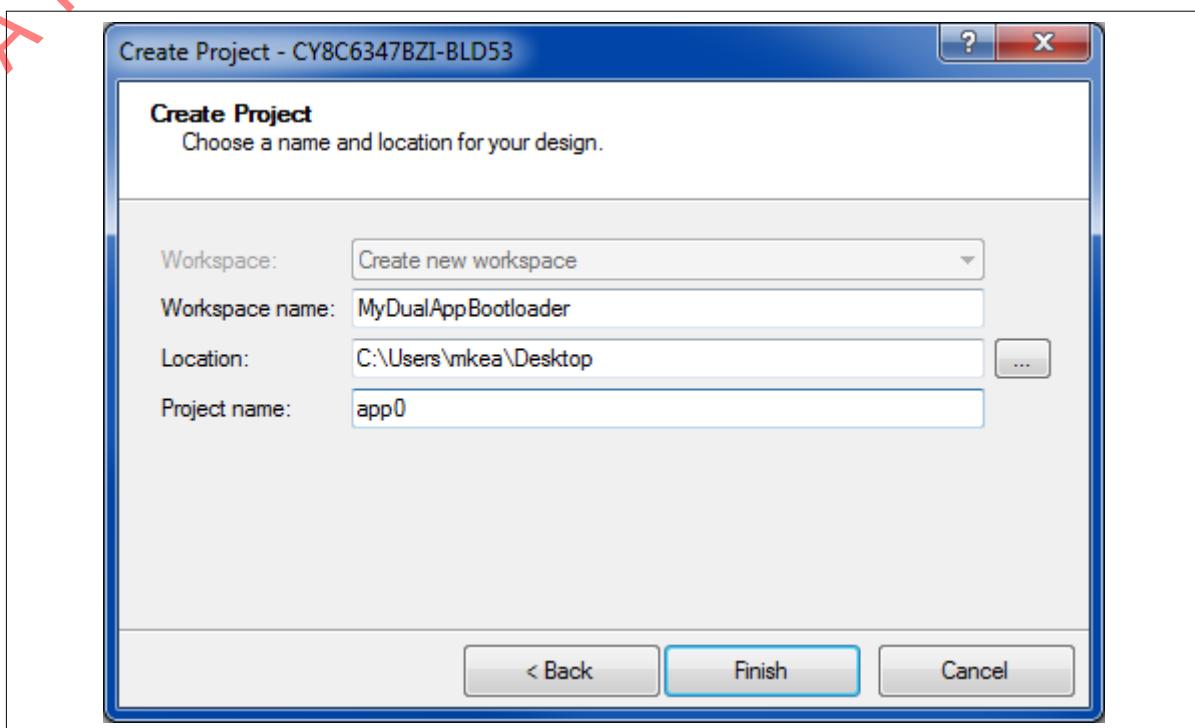


Figure 592 PSoC Creator create dual-application bootloader App

2. Configure App0 as a bootloader in the project Build Settings; see [Figure 573](#). Select I2C for the communication channel
3. Select Build > Generate Application. This creates the bootloader linker files, e.g., *bootload_cm4.ld*
4. Go back to the project Build Settings and set the Custom Linker Script for CM0+ and CM4 to the respective bootloader linker script files – see [Figure 575](#)
5. Add an I2C Component to the App0 schematic and configure the Component according to [Figure 581](#). At this time, you can also add the Pin Components used by [CE221984](#); the easiest way to do this is to copy and paste the Components from the code example schematic
6. In the Design-Wide Resources window, Pins tab, connect the Component pins to the appropriate physical pins
7. Update the files listed in [Table 142](#) with the content of the corresponding files in [CE221984](#). Add code as needed for your application

Table 142 CE221984 App0 files modified from the bootloader SDK default

File	Description of modification from the default
<i>bootload_common.ld</i>	Addition of memory regions for App2
<i>bootload_mdk_common.h</i>	
<i>bootload_user.h</i>	Set the CY_BOOTLOAD_GOLDEN_IMAGE_IDS macro to identify App1 as the factory default image To change the code example mode, change the CY_BOOTLOAD_OPT_GOLDEN_IMAGE macro to a nonzero value
<i>bootload_user.c</i>	Changed transport functions to I ² C from UART Added code to Cy_Bootload_WriteData() to do the factory default (“golden image”) check
<i>main_cm0p.c</i>	Addition of code to do the code example functions
<i>main_cm4.c</i>	

~~5 PSoC™ 6 application notes~~

- ~~DRAFT~~
8. Build the app0 project, and program it into the target kit. Test the application download (bootload) process. You can use the test steps listed in the Operation section of the [CE221984](#) document. Try changing the CY_BOOTLOAD_OPT_GOLDEN_IMAGE macro to a nonzero value and note the different behavior of the bootloader when you download App1
You can test the bootloader using the app1 and app2 projects already in the code example, or create your own applications, as the following step shows:
 9. Add an app1 project and an app2 project. You can configure both projects as installable applications. The detailed steps required are presented in [Section 5.2.2 Step 3](#) and [Step 4](#). Significant tasks are:
 - Update the project build settings:
 - Peripheral Driver Library > Bootloader SDK. Select only the **Core** box. ([Figure 573](#))
 - Linker custom scripts. ([Figure 575](#))
 - Post-build batch file ([Figure 576](#))
 - Add the post_build_core1.bat file to the project, in the Shared Files folder. See [Appendix E](#) for typical batch file content
 - Update the project schematic and design-wide resources files with the content of the corresponding files in [CE221984](#). Modify the design as needed for your application
 - Update the files listed in [Table 143](#) with the content of the corresponding files in CE221984. Add code as needed for your application

File	Description of modification from the default
bootload_common.1d	Addition of memory regions for App2
bootload_mdk_common.h	
bootload_cmp0.1d	Specify the memory regions and App ID as being for app1 or app2
bootload_cm4.1d	
bootload_cmp0.scat	
bootload_cm4.scat	
bootload_user.h	Set the CY_BOOTLOAD_GOLDEN_IMAGE_IDS macro to identify App1 as the factory default image; to change the code example mode, change the CY_BOOTLOAD_OPT_GOLDEN_IMAGE macro to a nonzero value
main_cm0p.c	Addition of code to do the code example functions
main_cm4.c	

5.16.5.5 PSoC™ 6 MCU encrypted bootloader

This section shows how to build a bootloading system with security features, as demonstrated in the code example [CE222802](#). This code example is similar to the [basic bootloaders code example](#), with additional features – signing and encryption – needed for secure bootloading. App0 is the bootloader; the downloadable App1 can be signed, encrypted, or both.

In this code example, the bootloader uses a UART communication channel. Changing to a different communication channel is simple; for more information, refer to one of the [previous](#) sections.

To build this code example, do the following steps. For more information on these steps, refer to [PSoC™ 6 MCU Basic Bootloaders](#) or [How to use the SDK](#).

Note: *Signing and encryption require keys. Files containing default keys are provided in the code example, but for your applications you will want to create new keys. The code example supports creating new*

5 PSoC™ 6 application notes

keys, but to do so you must first install OpenSSL and Python in your computer. See the code example document for details.

- DRAFT**
1. Create App0 and the workspace, as [Figure 593](#) shows

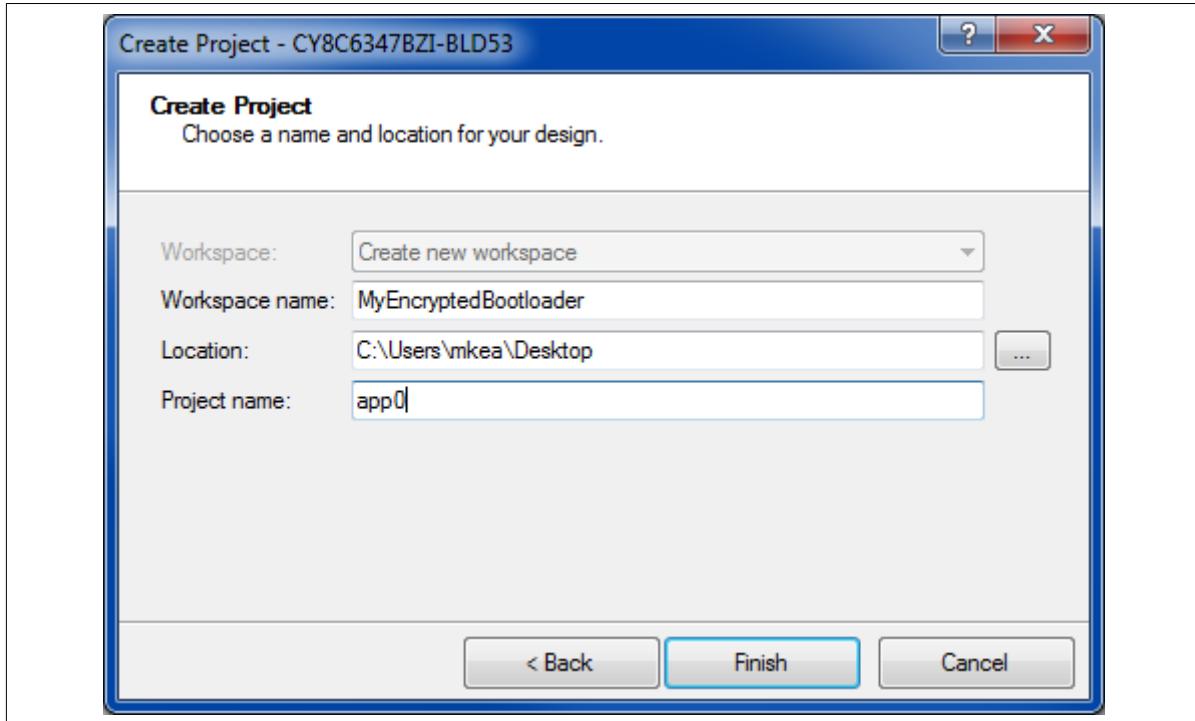


Figure 593 PSoC Creator create encrypted bootloader App0

2. Configure App0 as a bootloader in the project Build Settings; see [Figure 573](#). Select UART for the communication channel
3. Also in the Build Settings, select **Peripheral Driver Library > PDL > crypto**, and under Variant select Extra library to enable support for AES decryption
4. Select Build > Generate Application. This creates the bootloader linker files, e.g., *bootload_cm4.ld*
5. Go back to the project Build Settings and set the Custom Linker Script for CM0+ and CM4 to the respective bootloader linker script files; see [Figure 575](#). Also, add to the CM4 User Commands a call to the *post_build_core1.bat* file; see [Figure 576](#)
6. Add a UART Component to the App0 schematic and configure the Component according to [Figure 581](#). At this time, you can also add the Pin Components used by [CE222802](#); the easiest way to do this is to copy and paste the Components from the code example schematic
7. In the Design-Wide Resources window, Pins tab, connect the Component pins to the appropriate physical pins. Use [CE222802](#) and the guide for your kit for instruction
8. Update the files, and copy the new files, listed in [Table 143](#) from the corresponding files in [CE222802](#). Add code as needed for your application

Table 143 CE222802 App0 files modified from the bootloader SDK default

File	Description of modification from the default
<i>bootload_common.ld</i>	Different signature size
<i>bootload_mdk_common.h</i>	
<i>bootload_user.h</i>	Set #defines to enable the crypto block and secure verification
<i>bootload_user.c</i>	Addition of key variables and decrypt function

(table continues...)

~~5 PSoC™ 6 application notes~~

~~Table 143~~

(continued) CE222802 App0 files modified from the bootloader SDK default

File	Description of modification from the default
main_cm0p.c	Addition of code to do the code example functions
main_cm4.c	

New files; copy from the code example, and add to the Shared Files folder

cy_si_config.h	Key storage
cy_si_keystorage.h, .c	
post_build_core1.bat	Signs and encrypts the output file

9. Copy from [CE222802](#) the project folder CE222802_Keys.cylib into your workspace folder. This folder contains key files that are used by post_build_core1.bat. You can replace these keys with your own keys; see the code example document for instructions
10. The folder also contains a PSoC™ Creator library project file CE222802_Keys.cylib. You can optionally add this library project to your workspace
11. Build the app0 project, and program it into the target kit. Test the application download (bootload) process. You can use the test steps listed in the Operation section of the code example document

Note: *The bootloader project is built in a format such that it is validated by a Flash Boot module in PSoC™ 6 MCU supervisory flash (Sflash), as part of device initialization. If you reprogram the device to replace app0 with another project, validation will fail and the device will not boot. If you want to reprogram the device with other projects, you must first disable validation of app0 and then reprogram it into the device – see the instructions in the code example document.*

You can test the bootloader using the app1 project already in the code example, or create your own application, as the following step shows:

12. Add an app1 project. You can configure it as an installable application. The detailed steps required are presented in [Section 5.2.2 Step 3](#) and [Step 4](#). Significant tasks are:
 - Update the project build settings:
 - Peripheral Driver Library > Bootloader SDK. Select only the **Core** box. ([Figure 573](#))
 - Linker custom scripts. ([Figure 575](#))
 - Post-build batch file ([Figure 576](#))
 - Update the project schematic and design-wide resources files with the content of the corresponding files in [CE222802](#). Modify the design as needed for your application
 - Update the files, and copy the new files, listed in [Table 144](#) from the corresponding files in CE222802. Add code as needed for your application

Table 144

CE221984 App1 files modified from the bootloader SDK default

File	Description of modification from the default
bootload_common.ld	Specify memory regions
bootload_mdk_common.h	
bootload_cmp0.ld	Specify memory regions and App ID
bootload_cm4.ld	
bootload_cmp0.scat	
bootload_cm4.scat	
bootload_user.h	Set #defines to enable the crypto block and secure verification

(table continues...)

5 PSoC™ 6 application notes**Table 144****(continued) CE221984 App1 files modified from the bootloader SDK default**

File	Description of modification from the default
main_cm0p.c	Addition of code to do the code example functions
main_cm4.c	
New files; copy from the code example, and add to the Shared Files folder	
cy_si_config.h	Key storage
post_build_core1.bat	Signs and encrypts the output file

5 PSoC™ 6 application notes

References

-
- 1
- 6

For a comprehensive list of PSoC™ 6 MCU resources, see [KBA223067](#) in the Cypress community.

Application notes

AN221774 – Getting Started with PSoC™ 6 MCU	Describes PSoC™ 6 MCU devices and how to build your first ModusToolbox or PSoC™ Creator project
AN210781 – Getting Started with PSoC™ 6 MCU with Bluetooth Low Energy (BLE) Connectivity	Describes PSoC™ 6 MCU with BLE Connectivity devices and how to build your first PSoC™ Creator project
AN215656 – PSoC™ 6 MCU: Dual-CPU System Design	Describes the dual-CPU architecture in PSoC™ 6 MCU, and shows how to build a simple dual-CPU design
AN219434 – Importing PSoC™ Creator Code into an IDE for a PSoC™ 6 MCU Project	Describes how to import the code generated by PSoC™ Creator into your preferred IDE

Code examples

CE213903 – PSoC™ 6 MCU Basic DFU	Describes UART, I ² C, and SPI DFU for PSoC™ 6 MCU
CE216767 – PSoC™ 6 MCU with Bluetooth Low Energy (BLE) Connectivity Bootloader	Describes a BLE bootloader for PSoC™ 6 MCU
CE220959 – PSoC™ 6 MCU BLE Bootloader with External Memory	Similar to the BLE bootloader; the downloaded application is temporarily saved in external memory and then copied to its final destination
CE220960 – PSoC™ 6 MCU BLE Bootloader with Upgradeable Stack	Similar to the BLE bootloader; the BLE stack can be updated in addition to the application
CE221984 – PSoC™ 6 MCU Dual-Application I ² C Bootloader	Similar to the basic I ² C bootloader; manages two downloaded applications instead of one
CE222802 – PSoC™ 6 MCU Encrypted Bootloader	Similar to the basic UART bootloader; the application is digitally signed and encrypted

PSoC™ Creator component datasheets

UART	Supports UART-based communications
I ² C	Supports I ² C-based communications
SPI	Supports SPI-based communications
BLE	Supports BLE communications

Device documentation

PSoC™ 6 MCU: PSoC™ 63 with BLE Datasheet	PSoC™ 6 MCU: PSoC™ 63 with BLE Architecture Technical Reference Manual
--	--

Development kit documentation

CY8CKIT-062-BLE	PSoC 6 BLE Pioneer Kit
CY8CKIT-062-WiFi-BT	PSoC 6 WiFi-BT Pioneer Kit
CY8CPROTO-063-BLE	PSoC 6 BLE Prototyping Kit
CY8CPROTO-062-4343W	PSoC 6 Wi-Fi Prototyping Kit

5 PSoC™ 6 application notes

Tool documentation

ModusToolbox™	ModusToolbox IDE simplifies development for IoT designers. It delivers easy-to-use tools and a familiar microcontroller (MCU) integrated development environment (IDE) for Windows, macOS, and Linux.
PSoC™ Creator	PSoC Creator enables concurrent hardware and firmware editing, compiling and debugging of PSoC™ devices. Applications are created using schematic capture and over 150 pre-verified, production-ready peripheral Components. Look in the downloads tab for Quick Start and User Guides.
Peripheral Driver Library (PDL)	Installed by PSoC™ Creator 4.2. Look in the <PDL install folder>/doc for the User Guide and the API Reference

~~DATA~~ 5 PSoC™ 6 application notes

A DFU host tool

The DFU Host Tool (DHT) is a standalone graphical tool provided with ModusToolbox IDE. (There is also a Bootloader Host Program (BHP) provided with PSoC™ Creator. It operates much the same as DHT.) This tool is used to communicate with a PSoC™ device that has DFU installed. Using this tool, you can:

- Download and install an application to a device
- Verify an application that is already installed on a device
- Erase an application from a device

Note: *You cannot use DHT to install a DFU application into a device. Instead, you must program it through the device SWD/JTAG port, using other tools such as ModusToolbox IDE or Cypress Programmer. After a DFU application is installed, you can use DHT to install a downloadable application.*

DHT supports communicating with Cypress MCU devices via UART, I²C, SPI, or USB, as Figure 594 shows. You can see all devices available for connection. For UART or USB, communication can be done directly from your computer by connecting an appropriate cable. For I²C and SPI, a special communication port is needed, such as a KitProg module. The port configuration fields change depending on the selected port.

Notes:

1. At this time, only the UART, I²C, and SPI-based DFU are supported for PSoC™ 6 MCU.
2. DHT does not support Bluetooth Low-Energy (BLE). Use Cypress' CySmart product instead; see the CySmart User Guide section "Updating Peripheral Device Firmware".

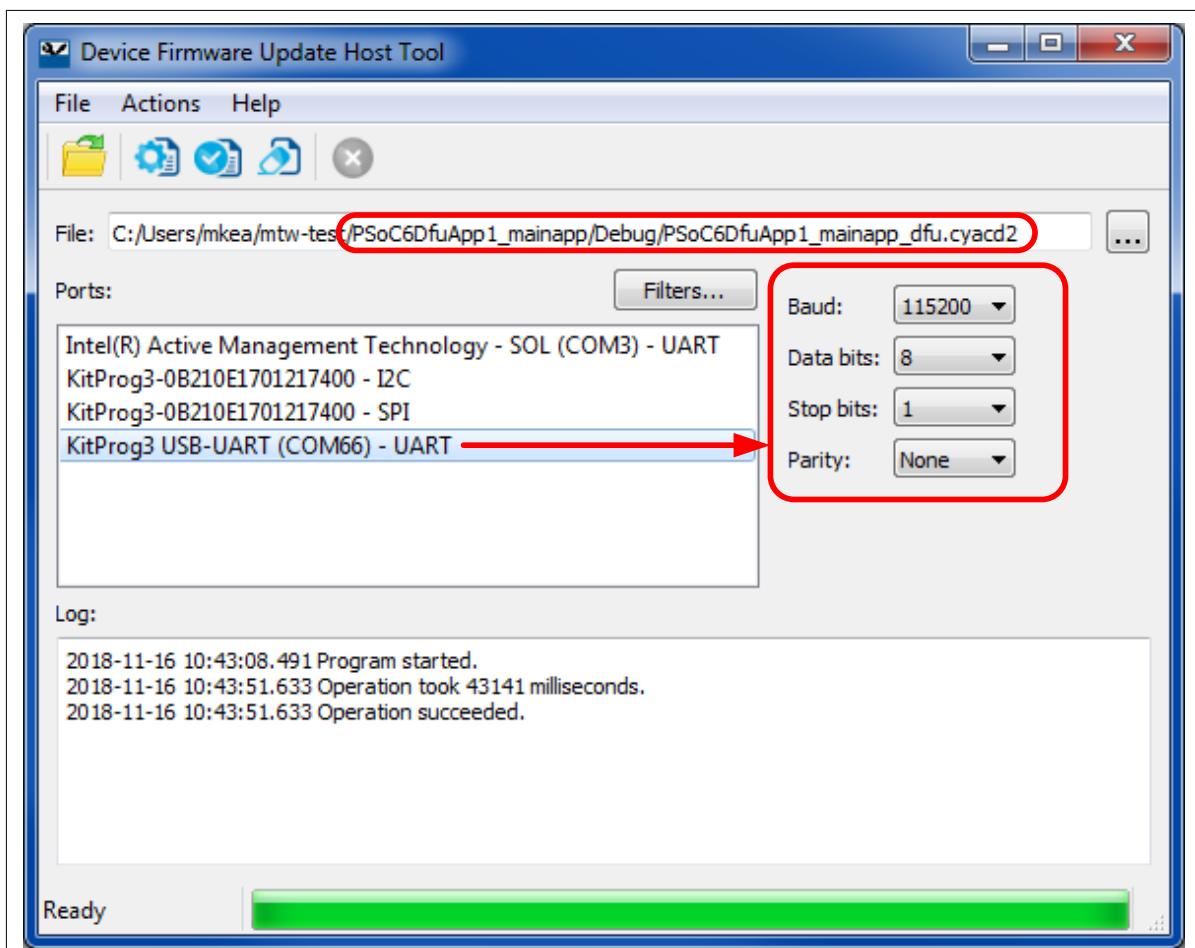


Figure 594 DFU host program graphical user interface (GUI)

~~5 PSoC™ 6 application notes~~

~~B Host command/response protocol~~

The DFU module communicates with a host using a simple command-response protocol, regardless of the communication channel used. The DFU module receives commands from the communication channel and responds to each command by sending one or more bytes to the communication channel. See [Figure 565](#). The commands and responses are in the form of a byte stream, packetized in a manner that ensures the integrity of the data being transmitted. A packet validity check method is included and consists of a 2's complement 16-bit checksum.

B.1 Command/Response packet structure

Communication packets sent from the host to the DFU module have the structure shown in [Figure 595](#):

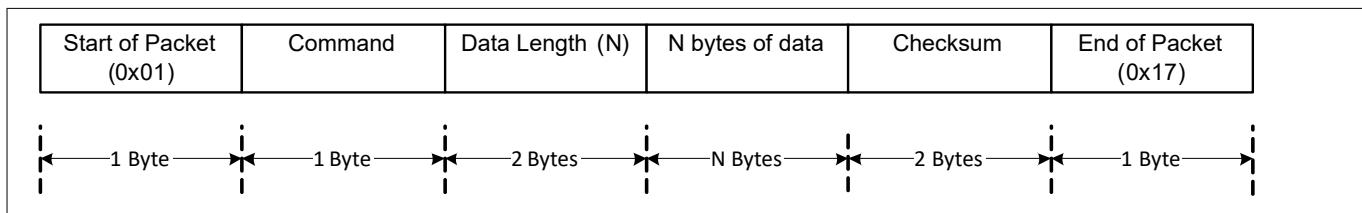


Figure 595 DFU command packet structure DFU response packet structure

Response packets sent from the DFU module to the host have the structure shown in [Figure 596](#):

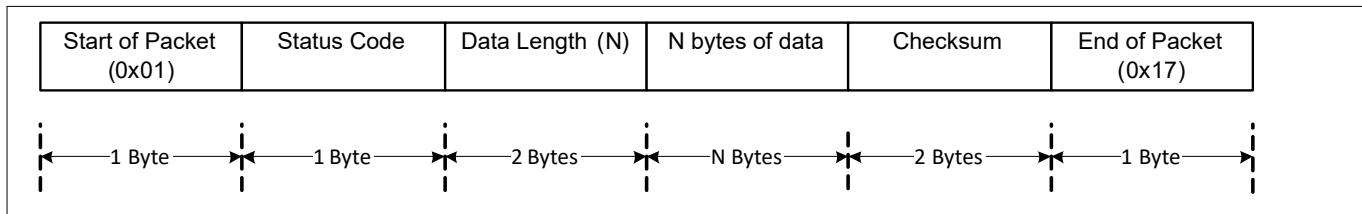


Figure 596 DFU response packet Structure

All multi-byte fields are LSB first.

B.2 Commands

[Table 145](#) shows a list of all commands supported by the DFU SDK. All commands except Exit DFU are ignored until the Enter DFU command is received.

Table 145 DFU commands list

DFU commands		
Enter/exit	DFU operation	Miscellaneous
Enter DFU	Send Data	Verify Application
Sync DFU	Send Data Without Response	Set Application Metadata
Exit DFU	Program Data	Get Metadata
	Verify Data	Set EIVector
	Appendix B.2.8	

There is no specific requirement for command execution time.

[Table 146](#) shows a list of all status and error codes supported by the DFU SDK.

~~5 PSoC™ 6 application notes~~

~~QAF~~ **Table 146 DFU status and error codes list**

Status/error code	Value	Description
CY_DFU_SUCCESS	0x00	The command was successfully received and executed
CY_DFU_ERROR_VERIFY	0x02	Verification of non-volatile memory (NVM) after writing failed
CY_DFU_ERROR_LENGTH	0x03	The amount of data sent is greater than expected
CY_DFU_ERROR_DATA	0x04	Packet data is not of the proper form
CY_DFU_ERROR_CMD	0x05	The command is not recognized
CY_DFU_ERROR_CHECKSUM	0x08	Packet checksum or CRC does not match the expected value
CY_DFU_ERROR_ROW	0x0A	The flash row number is not valid
CY_DFU_ERROR_ROW_ACCESS	0x0B	The flash row number cannot be accessed, for example due to MPU protection
CY_DFU_ERROR_UNKNOWN	0x0F	An unknown error occurred

B.2.1 Enter DFU

Begins a DFU operation. All other commands except Exit DFU are ignored until this command is received. Responds with device information and DFU SDK version.

- Input
 - Command Byte: 0x38
 - Data Bytes:
 - - 4 bytes (optional): product ID. If these bytes are included, and they are not 00 00 00 00, they are compared to device product ID data.
- Output
 - Status/Error Codes:
 - Success
 - Error Command
 - Error Data, used for product ID mismatch
 - Error Length
 - Error Checksum
 - Data Bytes:
 - 4 bytes: Device JTAG ID
 - 1 byte: Device revision
 - 3 bytes: DFU SDK version

B.2.2 Sync DFU

Resets the DFU to a known state, making it ready to accept a new command. Any data that was buffered is discarded. This command is needed only if the DFU module and the host get out of sync with each other.

- Input
 - Command Byte: 0x35
 - Data Bytes: N/A
- Output: N/A – this command is not acknowledged

5 PSoC™ 6 application notes

~~DRAFT~~ B.2.3 Exit DFU

Exits from the DFU. Ends the DFU operation.

- Input
 - Command Byte: 0x3B
 - Data Bytes: N/A
- Output: N/A – This command is not acknowledged

B.2.4 Send data

Transfers a block of data to the DFU module. This data is buffered in anticipation of a Program Data or Verify Data command. If a sequence of multiple send data commands are sent, the data is appended to the previous block. This command is used to break up large data transfers into smaller pieces, to prevent channel starvation in some communication protocols.

- Input
 - Command Byte: 0x37
 - Data Bytes:
 - n bytes: Data to write or verify
- Output
 - Status/Error Codes:
 - Success
 - Error Command
 - Error Data
 - Error Length
 - Error Checksum
 - Data Bytes: N/A

B.2.5 Send data without response

Same as the Send Data command, except that no response is generated by the DFU module. This reduces DFU time for some applications.

- Input
 - Command Byte: 0x47
 - Data Bytes:
 - n bytes: Data to write or verify
- Output: N/A

B.2.6 Program data

Writes data to one row of the device internal flash or page of external nonvolatile memory (NVM). May follow a series of [Send Data](#) or [Send Data Without Response](#) commands.

- Input
 - Command Byte: 0x49
 - Data Bytes:
 - 4 bytes: Address. Must be within the correct memory address space, and appropriately aligned.
For internal flash, it must be aligned to a flash row boundary. For external memory, it must conform to external memory alignment requirements.

~~5 PSoC™ 6 application notes~~

- ~~DRAFT~~
- 4 bytes: CRC-32C of the entire data to be written. The data is verified both before and after programming.
 - n bytes: Data to write into the flash row or external NVM page.
 - Output
 - Status/Error Codes:
 - Success
 - Error Command
 - Error Data
 - Error Length
 - Error Checksum
 - Error Flash Row
 - Error Flash Row Access
 - Data Bytes: N/A

B.2.7 Verify data

Compares data to one row of the device internal flash or page of SMIF. May follow a series of [Send Data](#) or [Send Data Without Response](#) commands.

This command is optional; its presence depends on a user configuration macro in dfu_user.h.

- Input
 - Command Byte: 0x4A
 - Data Bytes:
 - 4 bytes: Address. Must be within the correct memory address space, and appropriately aligned. For internal flash, it must be aligned to a flash row boundary. For external memory, it must conform to external memory alignment requirements.
 - 4 bytes: CRC-32C of the entire data to be verified.
 - n bytes: Data to compare with the flash row or SMIF page.
- Output
 - Status/Error Codes:
 - Success
 - Error Verify
 - Error Command
 - Error Data
 - Error Length
 - Error Checksum
 - Error Flash Row
 - Error Flash Row Access
 - Data Bytes: N/A
- Implementation details
 - The command returns the “Success” status code if all data bytes match the bytes starting at the specified flash address, otherwise “Error Verify”.

B.2.8 Erase data

Erases the contents of the specified internal flash row or SMIF page.

~~5 PSoC™ 6 application notes~~

This command is optional; its presence depends on a user configuration macro in dfu_user.h.

- Input
 - Command Byte: 0x44
 - Data Bytes:
 - 4 bytes: Address. Must be within the correct memory address space, and appropriately aligned. For internal flash, it must be aligned to a flash row boundary. For external memory, it must conform to external memory alignment requirements.
- Output
 - Status/Error Codes:
 - Success
 - Error Command
 - Error Data
 - Error Length
 - Error Checksum
 - Error Flash Row
 - Error Flash Row Access
 - Data Bytes: N/A

B.2.9 Verify application

Reports whether the checksum for the application in flash or external NVM is valid.

- Input
 - Command Byte: 0x31
 - Data Bytes:
 - 1 byte: Application number of the application to be verified. May range from 0 to the number of applications minus one.
- Output
 - Status/Error Codes:
 - Success
 - Error Command
 - Error Data
 - Error Length
 - Error Checksum
 - Error Flash Row Access
 - Data Bytes:
 - 1 byte: 1/0 for application is valid or not valid

B.2.10 Set application metadata

This command is used to set a given application's metadata. See [Application Metadata](#).

Note: *This command does not update the metadata if the user configures the DFU SDK to keep the metadata unchanged.*

~~5 PSoC™ 6 application notes~~

- Input
 - Command Byte: 0x4C
 - Data Bytes:
 - 1 byte: Application #
 - 8 bytes: metadata field format per Appendix D
- Output
 - Status/Error Codes:
 - Success
 - Error Command
 - Error Data
 - Error Length
 - Error Checksum
 - Error Flash Row Access
 - Data Bytes: N/A

B.2.11 Get metadata

Reports selected metadata bytes.

This command is optional; its presence depends on a user configuration macro in dfu_user.h.

- Input
 - Command Byte: 0x3C
 - Data Bytes:
 - 2 bytes: from offset within row; 0 – 511
 - 2 bytes: to offset within row; 0 – 511 (inclusive)
- Output
 - Status/Error Codes:
 - Success
 - Error Command
 - Error Data
 - Error Length
 - Error Checksum
 - Error Flash Row Access
 - Data Bytes: N/A
 - N bytes – per from and to offset bytes (inclusive)

B.2.12 Set EIVector

Sets an encryption initialization vector (EIV). This enables the DFU module to decrypt data before writing it to flash.

This command is optional; its presence depends on a user configuration macro in dfu_user.h.

5 PSoC™ 6 application notes

- Input
 - Command Byte: 0x4D
 - Data Bytes:
 - n bytes: the vector; 0, 8, or 16 bytes, little-endian raw data
- Output
 - Status/Error Codes:
 - Success
 - Error Command
 - Error Length
 - Error Data
 - Error Checksum
 - Data Bytes: N/A

~~5 PSoC™ 6 application notes~~

~~C~~ .cyacd2 file format

The .cyacd2 file contains downloadable application data. It is created by CyMCUElfTool, and used by host programs such as Cypress' [DFU Host Tool](#) and [CySmart](#) to send applications to the target DFU module, as [Figure 568](#) shows. The file data is in the form of ASCII hex numbers, similar to Intel hex format. Each byte of data is represented by two characters. For example, a byte 0x1E is represented by the characters 0x31 (ASCII '1') followed 0x45 (ASCII 'E').

All multi-byte fields are little-endian.

The file consists of a series of lines, or rows. Each row is terminated with ASCII CR, LF characters. A row is one of the following types:

- Encryption initial vector: An encryption initial vector row is of the format @EIV:<bytes>. The data in <bytes> is used by the host program in the [Set EIVector](#) to the DFU module
- Application verification information: An application verification information row is of the format:

```
@APPINFO:[__cy_app_verify_start],[__cy_app_verify_length].
```

- The start and length data are used by the host program in the [Set Application Metadata](#) to the DFU module.
- Header: A header row has the structure shown in [Table 147](#).

Table 147 cyacd2 header row structure

File version	Silicon ID	Silicon revision	Checksum type	App ID	Product ID
1 byte	4 bytes	1 byte	1 byte	1 byte	4 Bytes

- File Version: Numbered starting at 1
- Silicon ID, Silicon Revision, Product ID: Used to prevent the application from being downloaded to the wrong device
- Checksum Type: The method used to verify a DFU packet (see [Command/Response Packet Structure](#)). 0 = checksum, 1 = CRC
- App ID: See [Figure 569](#). This also controls which portion of the application metadata is updated for this application
- Data: A data row has the structure shown in [Table 148](#)

Table 148 cyacd2 Data Row Structure

Header	Address	Data
1 character: ":"	4 bytes	N bytes

The value of N equals the total amount of data to be sent with a series of [Send Data](#) or [Send Data Without Response](#) commands followed by a [Program Data](#) or [Verify Data](#) command

The value of N typically, but not necessarily, equals the length of an NVM row. For example, if downloading into RAM, then N may be an arbitrary value.

~~5 PSoC™ 6 application notes~~

~~D Application metadata~~

The DFU SDK uses a designated region of NVM (or RAM in some cases) to store information about the applications – see [Figure 569](#). Metadata information is generally used for the following purposes:

- Validate an application
- Transfer control from one application to another
- Copy an application image from a temporary location to its designated location

As noted in [Figure 569](#), metadata typically occupies one flash row or NVM page. (In devices with small amounts of flash, multiple rows or pages may be used.) [Figure 569](#) also shows that metadata is located outside of any application.

[Table 149](#) contains symbols that are used to define the location, size and usage of the DFU metadata. The symbols are defined in the SDK linker script files and C source files.

Note: *All examples shown are for the GCC compiler and linker. Similar statements exist in source and linker script files for the MDK and IAR compilers and linkers.*

Table 149 **Metadata-related symbols**

Symbol	Defined in	Purpose
flash_boot_meta	dfu_cm0p.ld, dfu_cm4.ld	Defines the physical memory region that contains the metadata

Example usage: `flash_boot_meta (rw) : ORIGIN = 0x100FFA00, LENGTH = 0x400`

Defines a region in the last 1 KB of the 1-MB PSoC™ 6 MCU user flash (which starts at 0x1000 0000).

<code>__cy_boot_metadata_addr</code> <code>__cy_boot_metadata_length</code>	<code>dfu_cm0p.ld</code> , <code>dfu_cm4.ld</code>	These symbols define a compiler-independent memory address range that is used to store the metadata. These symbols are used in the DFU SDK code and may be used in user code.
--	---	---

```
Example usage: /* Bootloader SDK metadata limits */
/* Note that __cy_memory_0_row_size equals the row length in bytes of
   PSoC 6 user flash. */
__cy_boot_metadata_addr = ORIGIN (flash_boot_meta);
__cy_boot_metadata_length = __cy_memory_0_row_size;
```

<code>.cy_boot_metadata</code>	<code>dfu_cm0p.ld</code> , <code>dfu_cm4.ld</code>	The DFU metadata is stored in this section. At build time, CyMCUElfTool calculates the checksum of this section and places it in the last four bytes of the section. When a checksum is not needed, rename the section to any other name.
--------------------------------	---	---

```
Example usage: cy_boot_metadata :
{
    KEEP(*(.cy_boot_metadata))
} > flash_boot_meta
```

5 PSoC™ 6 application notes

Table 149 (continued) Metadata-related symbols

Symbol	Defined in	Purpose
CY_DFU_MAX_APPS	dfu_user.h	Allows the user to control the maximum number of applications supported in the DFU metadata

Example usage: /* The smallest metadata size is CY_DFU_MAX_APPS * 8 bytes per app +
an optional 4 bytes for metadata checksum
#define CY_DFU_MAX_APPS (2u)

CY_DFU_METADATA_WRITABLE	dfu_user.h	The DFU does not necessarily write metadata – it can be done by the application or some other user code. Or metadata may be set using a compile-time constant within an application. An application can have metadata that is smaller than an NVM row. In all these cases, set this macro to 0 to prevent the DFU from writing metadata. However, note that the DFU requires that a metadata region exist and be properly initialized. This may be done by the DFU itself, or by an application.
--------------------------	------------	--

Example usage: /* A non-zero value allows writing metadata with the SetAppMetadata command.
*/
#define CY_DFU_METADATA_WRITABLE (1)

5 PSoC™ 6 application notes

E Metadata structure

Application metadata has eight bytes of data per application, followed by four bytes for checksum, as [Table 150](#) shows:

Table 150 **Metadata structure**

Application 0	Application 1	...	Application N – 1	Checksum
8 bytes	8 bytes		8 bytes	4 bytes

Application Start Address	Application Length (bytes)
4 bytes	4 bytes

You can set the number of applications N in the dfu_user.h file:

```
#define CY_DFU_MAX_APPS (N)
```

The default value of N is 2.

Each application start address must be aligned to a flash row or NVM page boundary. The application length must be a multiple of the flash row or NVM page length.

The Checksum is calculated with the same algorithm that is used in the DFU commands [Program Data](#) and [Verify Data](#). The default algorithm is CRC-32C.

5 PSoC™ 6 application notes

F Post-build file listings

The following are listings of the post-build bash and batch files for App1, from [CE213903](#). The files are similar if not the same in the other code examples. The same bash/batch can be used with multiple downloadable applications, for example app2 in the [PSoC™ 6 MCU dual-application bootloader](#).

F.1 ModusToolbox™ post-build bash file

```
#!/bin/bash

#####
# This script is designed to post process a PSoC 6 application. It performs sign and merge.
#
# usage:
#   dfu_postbuild.bash <MCUELFTOOL_LOC> <CM0P_LOC> <CM4_LOC> <MCU_CORE>
#
#####

MCUELFTOOL_LOC=$1
CM0P_LOC=$2
CM4_LOC=$3
MCU_CORE=$4

echo Script: cymcuelftool_postbuild
echo 1: MCUELFTOOL_LOC : $MCUELFTOOL_LOC
echo 2: CM0P_LOC       : $CM0P_LOC
echo 3: CM4_LOC        : $CM4_LOC
echo 4: MCU_CORE       : $MCU_CORE
echo

filenameNoExt_cm0p="${CM0P_LOC%.*}"
filenameNoExt_cm4="${CM4_LOC%.*}"

if [ "$MCU_CORE" == "ARM_CM4" ]; then
$MCUELFTOOL_LOC --sign $CM4_LOC --output $filenameNoExt_cm4"_signed.elf"
$MCUELFTOOL_LOC --merge $filenameNoExt_cm4"_signed.elf" $filenameNoExt_cm0p"_signed.elf" --
output $filenameNoExt_cm4"_final.elf"

#DFU
$MCUELFTOOL_LOC --sign $filenameNoExt_cm4"_final.elf" CRC --output $filenameNoExt_cm4"_dfu.elf"
$MCUELFTOOL_LOC -P $filenameNoExt_cm4"_dfu.elf" --output $filenameNoExt_cm4"_dfu.cyacd2"

else
$MCUELFTOOL_LOC --sign $CM0P_LOC --output $filenameNoExt_cm0p"_signed.elf"
  if [ -e $filenameNoExt_cm4"_signed.elf" ]; then
    $MCUELFTOOL_LOC --merge $filenameNoExt_cm4"_signed.elf" $filenameNoExt_cm0p"_signed.elf" --
output $filenameNoExt_cm4"_final.elf"
  fi
fi
```

5 PSoC™ 6 application notes**F.2 PSoC™ Creator post-build batch file**

DRAFT

```
@rem Usage:  
@rem Call post_build_core1.bat <tool> <output_dir> <project_short_name>  
@rem E.g. in PSoC Creator 4.2:  
@rem     post_build_core1.bat creator ${OutputDir} ${ProjectShortName}  
  
@echo -----  
@echo Post-build commands for Cortex-M4 core  
@echo -----  
  
@rem Set proper path to your PDL 3.x and above installation  
@set PDL_PATH="C:\Program Files (x86)\Cypress\PDL\3.0.1"  
  
@set CY MCU ELF TOOL=%PDL_PATH%\tools\win\elf\cymcuelftool.exe"  
  
@set IDE=%1  
  
@if "%IDE%" == "creator" (  
    @set OUTPUT_DIR=%2  
    @set PRJ_NAME=%3  
    @set ELF_EXT=.elf  
)  
  
@if "%IDE%" == "uvision" (  
    @set OUTPUT_DIR=%2  
    @set PRJ_NAME=%3  
    @set ELF_EXT=.axf  
)  
  
@if "%IDE%" == "iar" (  
    @set OUTPUT_DIR=%2  
    @set PRJ_NAME=%3  
    @set ELF_EXT=.out  
)  
  
@if "%IDE%" == "eclipse" (  
    @set OUTPUT_DIR=%2  
    @set PRJ_NAME=%3  
    @set ELF_EXT=  
)  
  
%CY MCU ELF TOOL% -S %OUTPUT_DIR%\%PRJ_NAME%\%ELF_EXT% CRC  
%CY MCU ELF TOOL% -P %OUTPUT_DIR%\%PRJ_NAME%\%ELF_EXT% --output %OUTPUT_DIR%\%PRJ_NAME%.cyacd2
```

5 PSoC™ 6 application notes

5.16.12 Revision history

Document version	Date of release	Description of changes
**	2017-03-08	New Application Note.
*A	2017-07-10	Updated for release of PSoC™ Creator 4.1 and PDL 3.0.0. Removed an error code from section B.2.12. Added Appendix E. Miscellaneous edits throughout.
*B	2017-08-25	Updated for release of PSoC™ Creator 4.2 and PDL 3.0.1. Added support for I²C to basic bootloaders instructions. Other edits. Ported to new application note document template. Confidential tag removed.
*C	2018-01-04	Updated for release of PSoC™ Creator 4.2 ES100. Added support for SPI to basic bootloaders instructions. Added support for BLE bootloader and BLE bootloader with external memory. Added Table 149 to Appendix D. Other edits. Ported to new application note document template.
*D	2018-03-27	Added support for code examples CE220959 and CE221984. Added figures 17 and 18. Miscellaneous minor updates and edits throughout the document. Ported to new application note document template.
*E	2018-02-25	Added support for CE220960 and CE222802. Added instructions for application signature data, in sections Chapter 5.16.4.3 and PSoC™ 6 MCU basic DFUs .
*F	2018-12-08	Added support for ModusToolbox IDE and DFU SDK. Term “bootload” changed to “DFU”.
*G	2021-03-30	Migrated to Infineon template.
*H	2022-07-21	Template update

5.17 AN235691 ModusToolbox™ & Friends

About this document

-
- 1
- 7

Scope and purpose

This document guides you in learning how you can become a part of the [Infineon Partner Program](#) and integrate your software content into the [ModusToolbox™](#) ecosystem.

Intended audience

This document is intended for partners registered in the Infineon Partner Program and want to integrate their software content into the [ModusToolbox™](#) ecosystem.

Reference documents

See the following documents for more information as needed:

- [ModusToolbox™ user guide](#)
- [ModusToolbox™ runtime software](#)
- [Project Creator user guide](#)
- [Library Manager user guide](#)
- <https://github.com/Infineon>

5 PSoC™ 6 application notes~~DRAFT~~
5.17.1 Introduction

ModusToolbox™ & Friends is a development program which extends the increased productivity, and feature-rich platform of ModusToolbox™ with highly innovative, robust and product ready partner software for developers. By opening ModusToolbox™ to partners, it provides a simple and tested integration of valuable software to developers for easy evaluation and integration to meet their product needs. Each partner owns their own software license, allowing them to maintain control of their engagement model.

To get started with this program, you first need to be a part of the [Infineon Partner Network](#). See upcoming sections for more details.

5.17.1.1 What is a partner?

A partner is a company selected by Infineon based on their proven competence and ability to design and deliver strong and trustworthy solutions, especially for new technologies and application fields. This way the partners can be a huge value add to the ModusToolbox™ ecosystem through development kits / production boards, innovative software IP, middleware or system references to showcase their design skills. See [Partner Network](#) webpage for more information.

5.17.1.2 Why become a partner?

ModusToolbox™ provides a rich platform with support for multiple IDEs, toolchains, software tools and libraries, development kits, reference examples etc. making it the platform of choice for developers. By being a partner, you can leverage everything ModusToolbox™ has to offer and distribute your own software via ModusToolbox™ to market your product and services directly to developers. See [Partner Brochure](#) to learn more.

5.17.1.3 How to become a partner?

To become a part of the partner ecosystem, the process is fairly simple. Go to the [Infineon Partner Network](#) webpage and apply to become a partner. The partner management team will evaluate how you can contribute to the network and help with onboarding.

~~5 PSoC™ 6 application notes~~

~~DRAFT~~ 5.17.2 ModusToolbox™ software overview

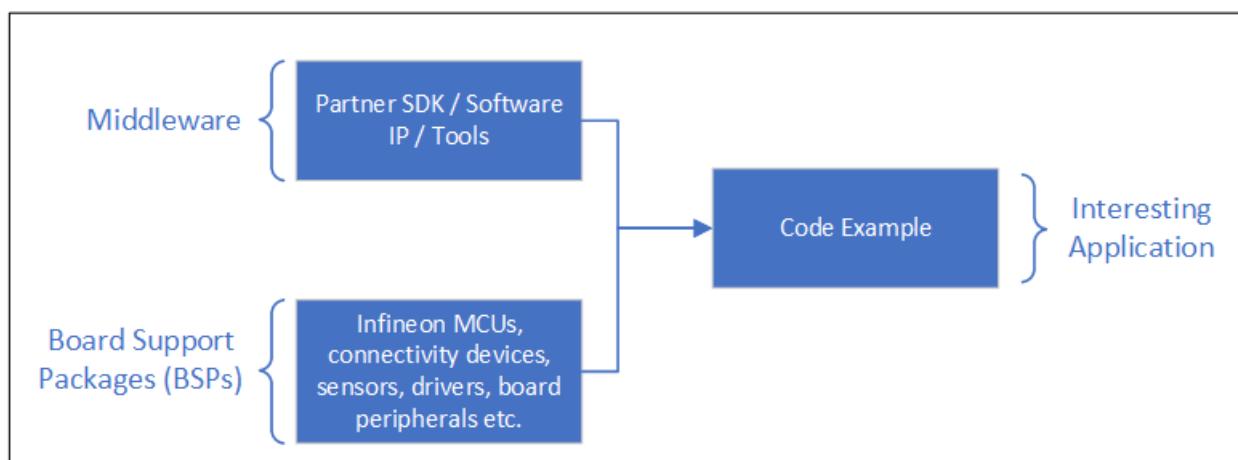
ModusToolbox™ software is a modern, extensible development environment supporting a wide range of Infineon microcontroller devices, wireless connectivity devices, and sensors. It provides a flexible set of tools and a diverse, high-quality collection of application-focused software. These include configuration tools, low-level drivers, libraries, and operating system support, most of which are compatible with Linux-, macOS-, and Windows-hosted environments. For more information, refer to the [ModusToolbox™ user guide](#).

5.17.2.1 Types of software content

ModusToolbox™ comprises mainly three types of software content that are hosted online using Git repositories (Infineon GitHub and third-party Git servers):

- Code examples
- Middleware
- Board Support Packages (BSPs)

As a partner, you can contribute to any or all of these software types. An illustration of how all these software types interact with each other is shown below:



In the upcoming sections, we discuss each of the software types in brief and the use-cases which mandate the creation of content in these categories. To understand more about these software types in detail, refer to the [ModusToolbox™ user guide](#) and [ModusToolbox™ run-time software reference guide](#).

5.17.2.1.1 Code examples

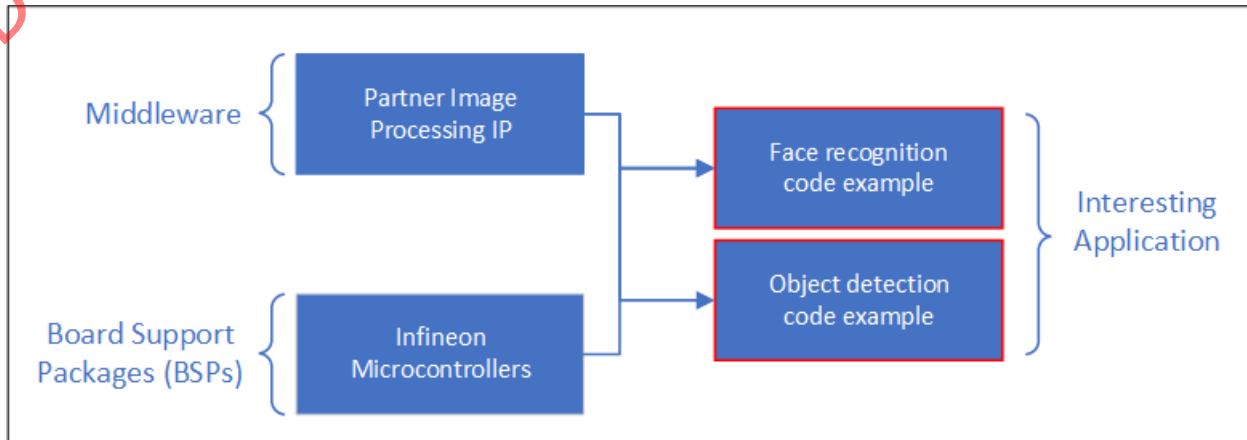
The code examples are ModusToolbox™ applications intended to demonstrate the usage of a particular product, feature, or tool. Users should be able to easily consume these examples by copying and pasting it into their source code.

All current ModusToolbox™ code examples can be found through the GitHub [code example page](#). There you will find links to examples for the Bluetooth® SDK, PSoC™ 6 MCU, and PSoC™ 4 device, among others. See [Creating a code example](#) for more information.

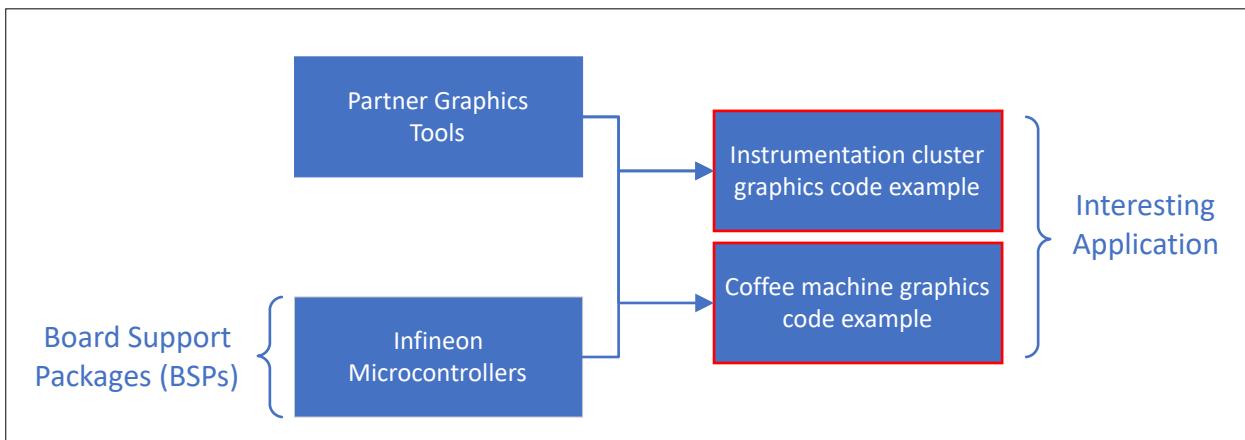
Let's look at a few demonstrative scenarios to understand when you would develop a code example.

~~5 PSoC™ 6 application notes~~

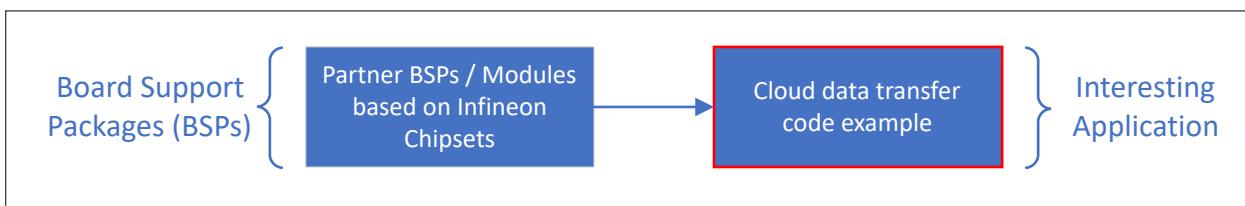
Scenario #1 – Demonstrate integrating your proprietary firmware SDK or software IP: For example, if you are a partner dealing with cutting edge image processing solutions, you would demonstrate how an Infineon microcontroller can be leveraged to interface with your solutions to create some interesting application like face recognition or object detection.



Scenario #2 – Demonstrate your tools to simplify processes: As a partner focused on cutting-edge tools to simplify a particular process, the usage of these tools would normally be explained in a code example. For example, a tool that simplifies creation of graphics on LCD displays may be demonstrated using a graphics-based code example with instructions on tool usage.



Scenario #3 – Demonstrate your modules: As a partner focused on creating modules or hardware using Infineon chipsets, a code example can be used to demonstrate how the custom module or evaluation kits (called custom BSP) can be interfaced to create interesting applications. For example, a partner focused on creating IoT prototyping modules using Infineon chipsets can develop a code example to demonstrate how to use their module to send data to the cloud.

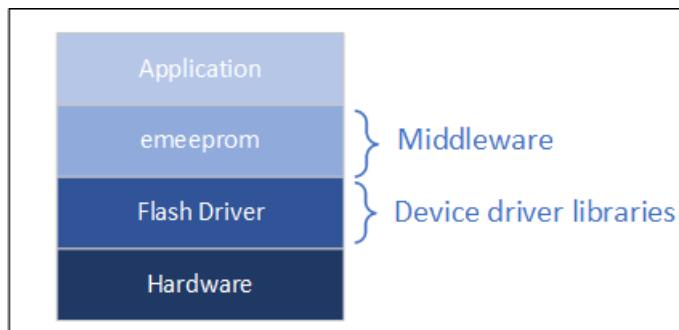


Note: These scenarios are not an exhaustive list and any number of combinations between the different software types is possible and permitted. If you are unsure where your content belongs, you can contact technical support to seek help. See section [Technical support](#) for more details.

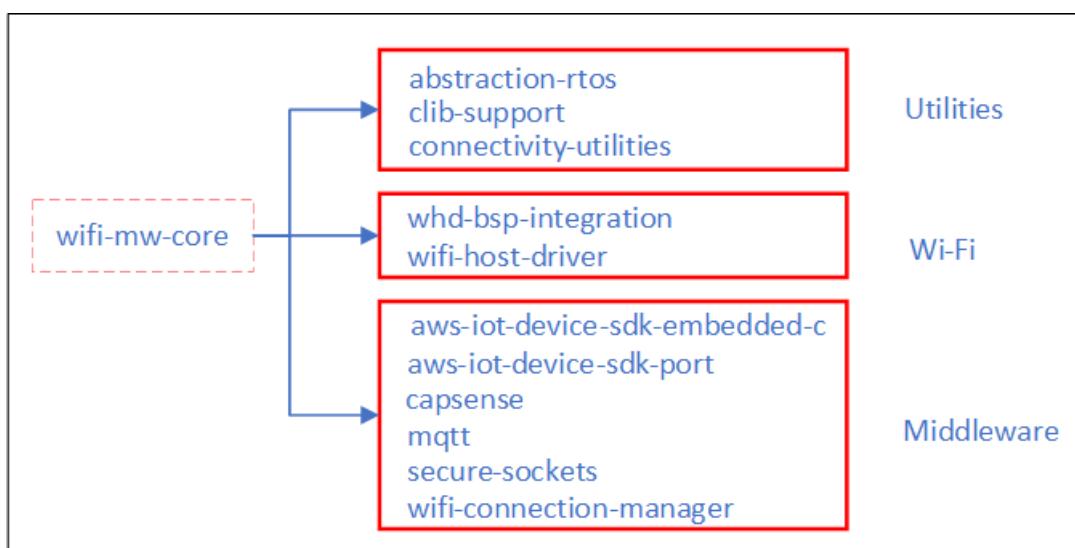
~~5 PSoC™ 6 application notes~~

~~5.17.2.1.2~~ Middleware

A middleware library is usually a wrapper on top of the low-level device driver libraries intended to demonstrate a particular feature or application by further abstracting the hardware level details from the user. For example, an Emulated EEPROM middleware library (emeeprom) operates on top of the Flash driver and allows users to easily store and manage data in an emulated EEPROM memory region.



There could be dependencies between several middleware libraries. For example, wifi-mw-core depends on other libraries like abstraction-rtos, clib-support etc. as shown below to work correctly.

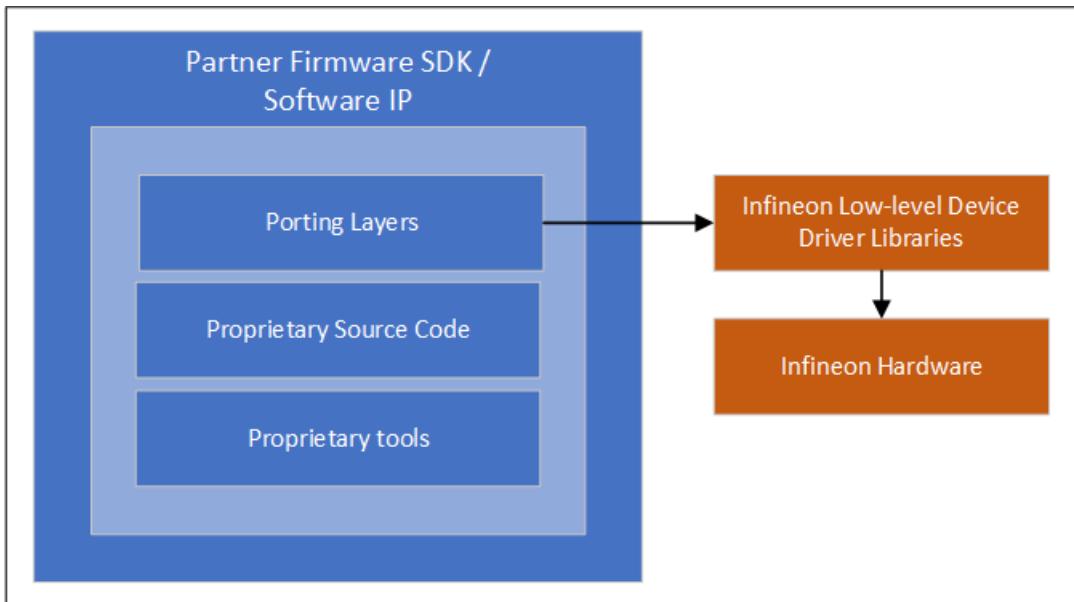


More information on creating your own middleware can be found in Section [Creating a middleware library](#). Let's look at a few scenarios to understand when you would develop a middleware library.

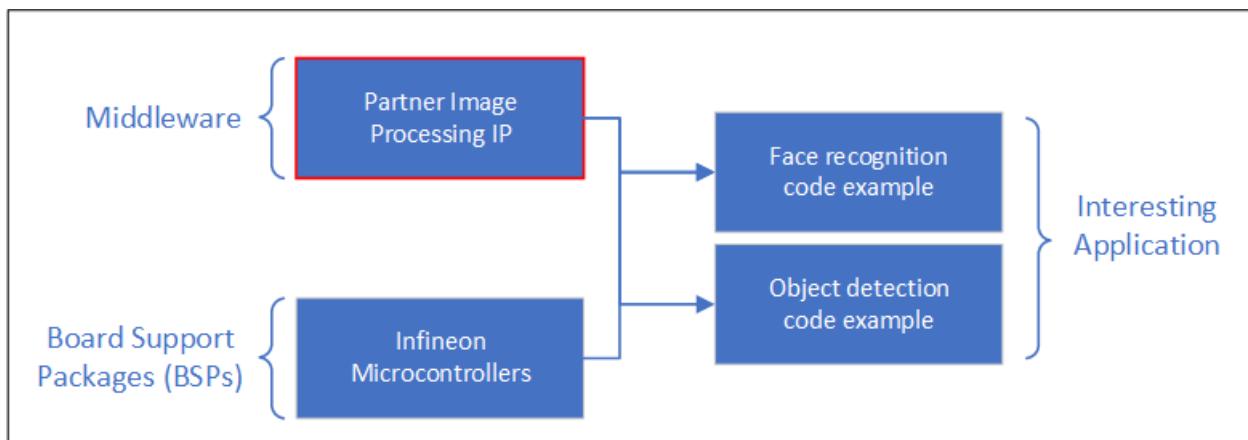
5 PSoC™ 6 application notes

~~DRAFT~~

Scenario #1 - As a partner, your proprietary firmware SDK or software IP usually belongs in the middleware category. To support creation of applications using your firmware SDK or software IP on Infineon microcontrollers, some modifications might be necessary to create porting layers as shown below.



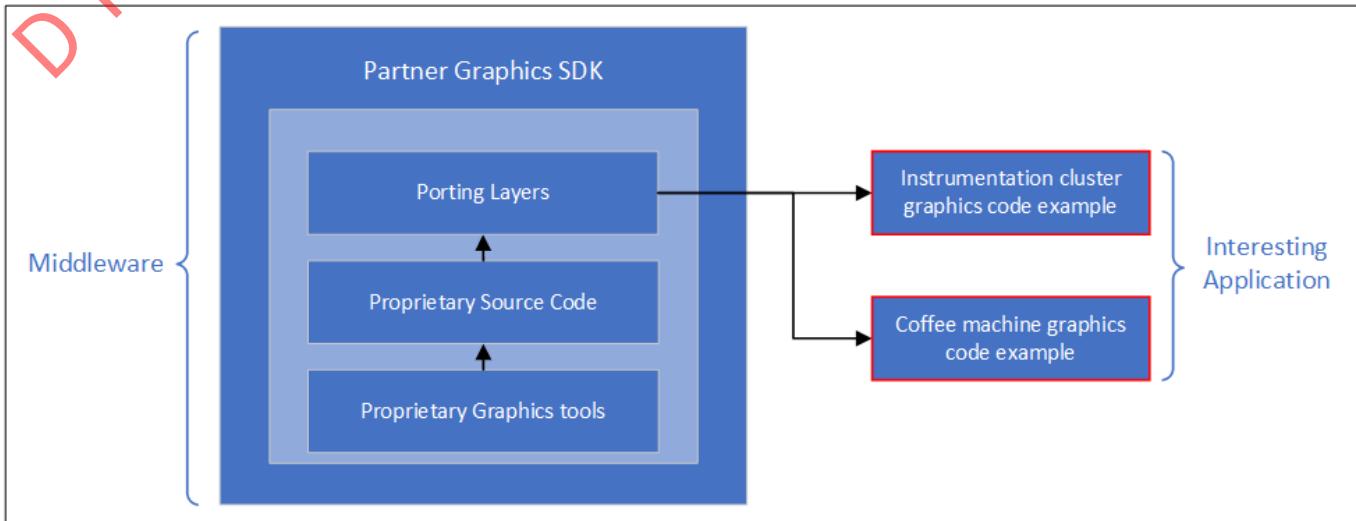
This modified firmware SDK or software IP with the porting layers is then offered as a middleware library that users can leverage in their code examples to interface with Infineon microcontrollers to create interesting applications.



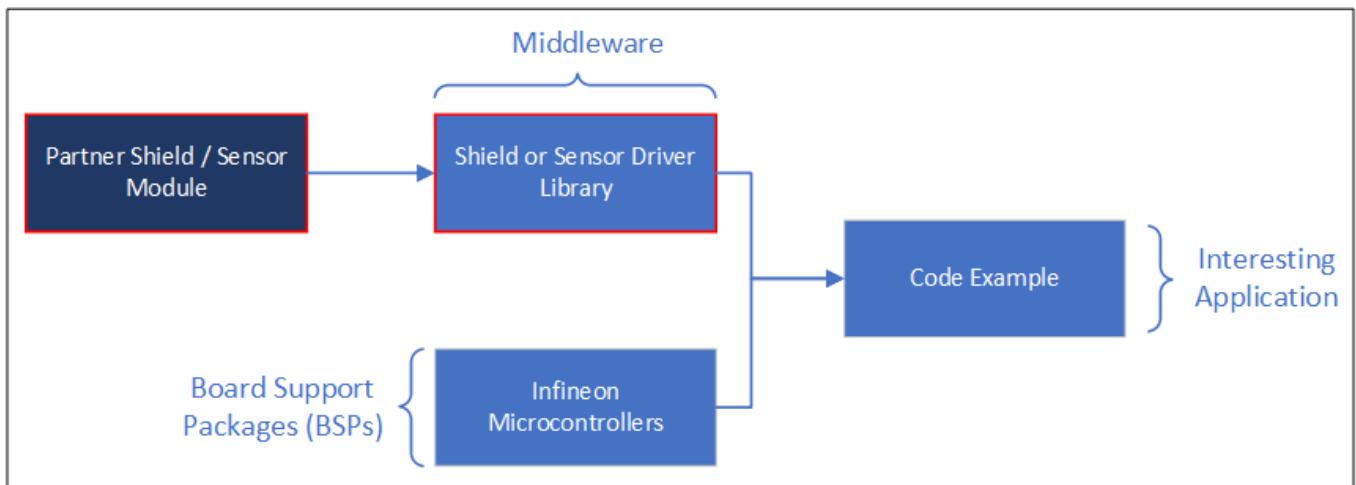
5 PSoC™ 6 application notes

~~DRAFT~~

Scenario #2 - If the tools require a particular IP or an SDK to work correctly, the tools should be offered as part of the middleware and not directly as a code example. For example, a partner focused on graphics tools that rely on a software SDK must bundle the SDK and tools together as shown below.



Scenario #3 - As a partner offering shields or sensors, the source code for interfacing with the shield or sensor should be provided as a middleware library. For example, a partner manufacturing a pressure sensor will develop the middleware library that allows users to interact with their sensor module easily using a simple API. This allows users to easily interface with any Infineon kit and obtain pressure data.



Reference examples that fall under this use case:

- [CY8CKIT-028-EPD](#) – This middleware library from Infineon provides the pin mapping and APIs to easily interface with the E-ink display shield board.
- [BMI160 Sensor Driver](#) – This middleware library from Bosch provides APIs to interface with the BMI160 inertial measurement unit (IMU) sensor.

Note: *These use-cases are not an exhaustive list and any number of combinations between the different software types is possible and permitted. If you are unsure where your content belongs, you can contact technical support to seek help. See section [Technical support](#) for more details.*

~~DEAF~~ 5 PSoC™ 6 application notes

5.17.2.1.3 Board support packages (BSPs)

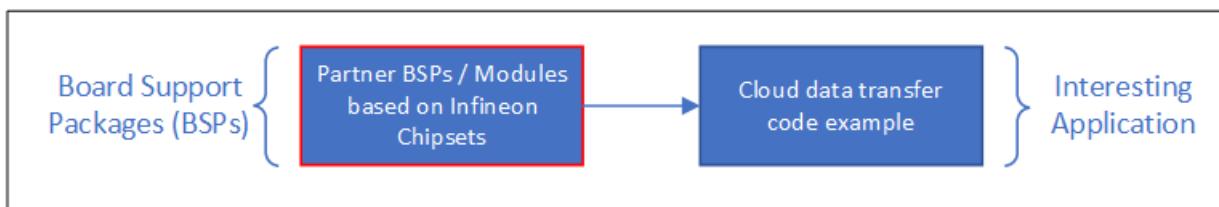
The Board Support Package (BSP) is intended to specify critical hardware items needed by the application, including:

- Hardware configuration files for the device (for example, design.modus)
- Startup code and linker files for the device
- Other libraries that are required to support a kit

This section is intended for partners who offer custom modules or hardware based on Infineon chipsets. For partners focusing on only software solutions/services, you can leverage the existing BSPs offered by Infineon.

Let's look at a demonstrative scenario to understand when you would develop a BSP:

As a partner focused on creating modules or hardware using Infineon chipsets, a code example can be used to demonstrate how the custom module or evaluation kits (called custom BSPs) can be interfaced to create interesting applications. For example, a partner focused on creating IoT prototyping modules using Infineon chipsets can develop a BSP to simplify the development of applications targeting their modules or evaluation kits.



Note: *These scenarios are not an exhaustive list and any number of combinations between the different software types is possible and permitted. If you are unsure where your content belongs, you can contact technical support to seek help. See section [Technical support](#) for more details.*

5.17.2.2 Manifests

ModusToolbox™ uses the concept of *manifests* to load the content to be displayed in its tools (Project Creator and Library Manager). Manifests are XML formatted files that contain a list of URLs that point to the appropriate libraries. The tools discover the locations of these Git repositories by loading the manifests hosted from preconfigured locations in Infineon GitHub.

There are multiple types of manifest files:

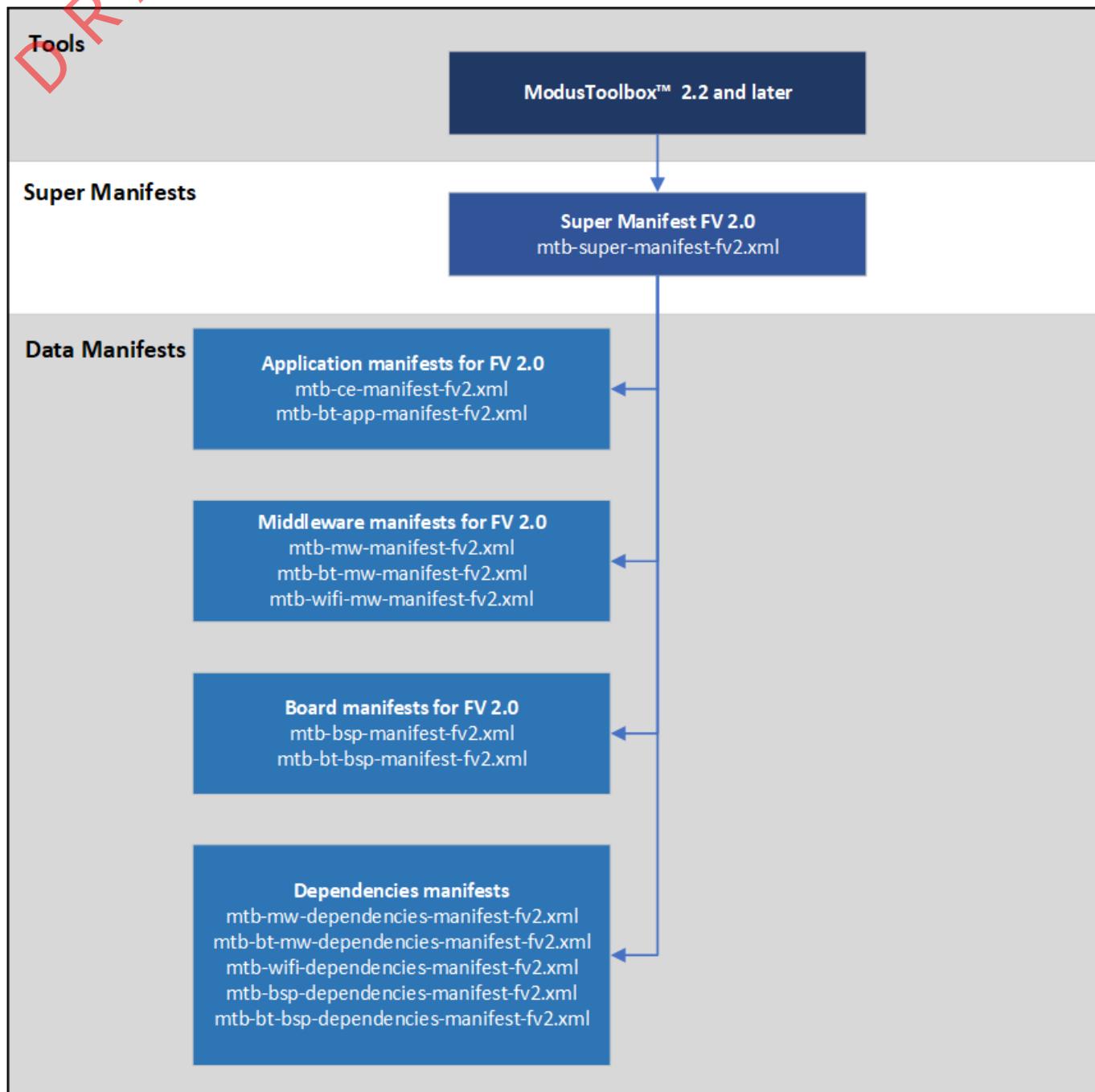
- The “super manifest” file contains a list of URLs that software uses to find the board, code example, and middleware manifest files.
- An “app manifest” file contains a list of code examples that should be made available to the user.
- A “board manifest” file contains a list of boards that should be 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.
- A “middleware manifest” file contains a list of available middleware libraries. There is also a separate middleware dependencies manifest that lists the dependent libraries associated with each middleware library.

The super manifest can list any number of app, board, and middleware manifest files. Each entry in an app, board, or middleware manifest file can contain more than one version of each library if desired. This allows new versions to be released while still allowing existing users to use any version they desire for their application needs.

Beginning with the ModusToolbox™ 2.2 release, there are two versions of manifest files: the old versions for the ‘LIB’ flow used with ModusToolbox™ 2.1 and earlier, and new versions for the ‘MTB’ flow (aka “fv2”), which is

5 PSoC™ 6 application notes

used with ModusToolbox™ 2.2 and later. In this application note, we will only cover the new “fv2” version of the manifests and content developed using ModusToolbox™ 2.2 and later.



Partners can set up a similar infrastructure of manifests to point to content and showcase it within ModusToolbox™ tools (Project Creator and Library Manager). See upcoming sections to learn more about how you can set up such an infrastructure.

~~DEAF~~ 5 PSoC™ 6 application notes

5.17.2.3 Git versioning control system

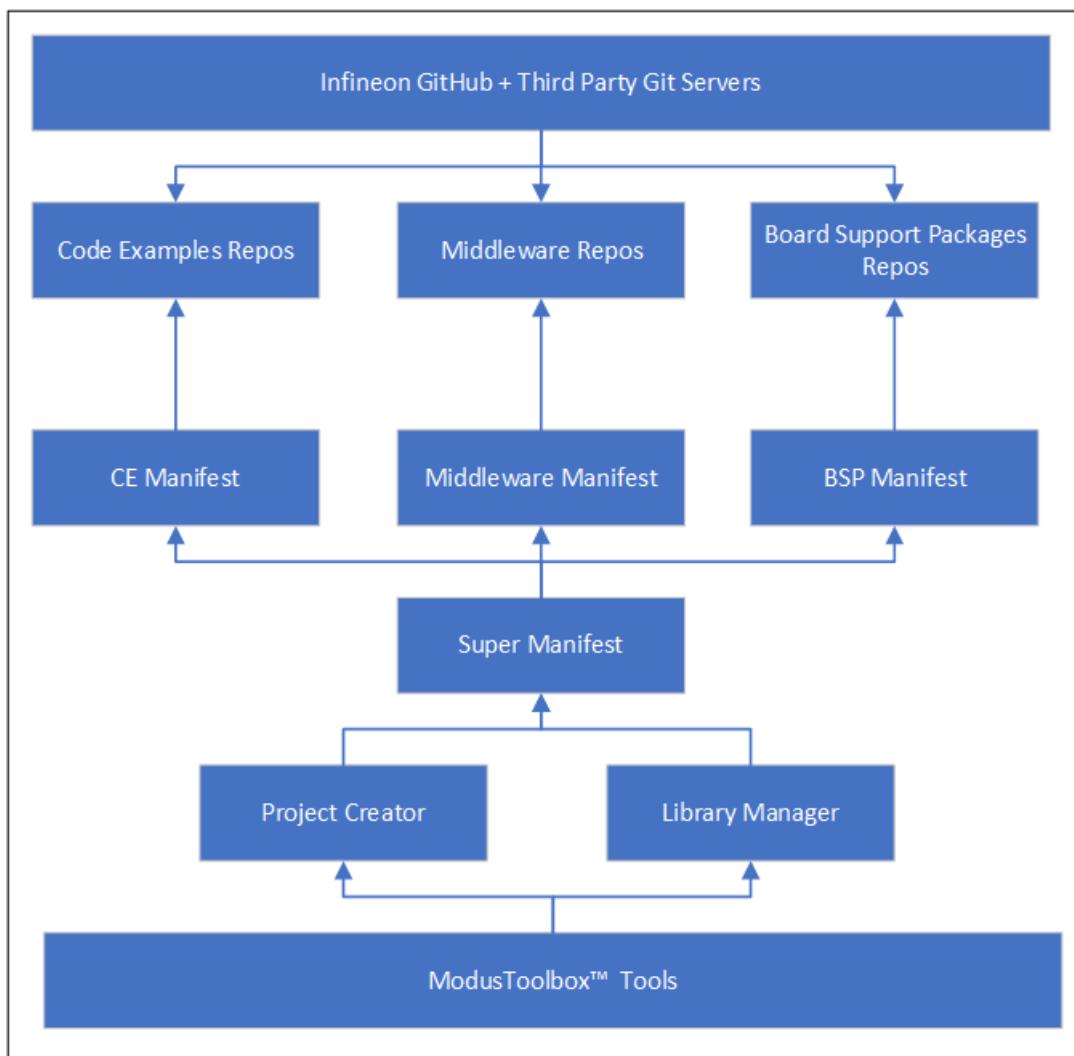
The Git versioning control system is central to how software content in ModusToolbox™ is offered. All the software content is hosted on Git repositories (Infineon GitHub and third-party Git servers). This allows content to be released regularly without requiring any update to the ModusToolbox™ tools.

As discussed previously, the manifests point to these Git repositories to allow discovery of the content from the tools. The tools run Git commands in the background to fetch the content from Git into your development environment. This is why it is important to host content on version control platforms based on Git so that the tools can work with them seamlessly.

The software content contains release tags that allow a specific version of the library to be brought in to the development environment. Because the software content is version-controlled, if any issues occur in a newly released version, it can be easily mitigated by rolling back to the last working version.

5.17.2.4 How does everything come together in ModusToolbox™?

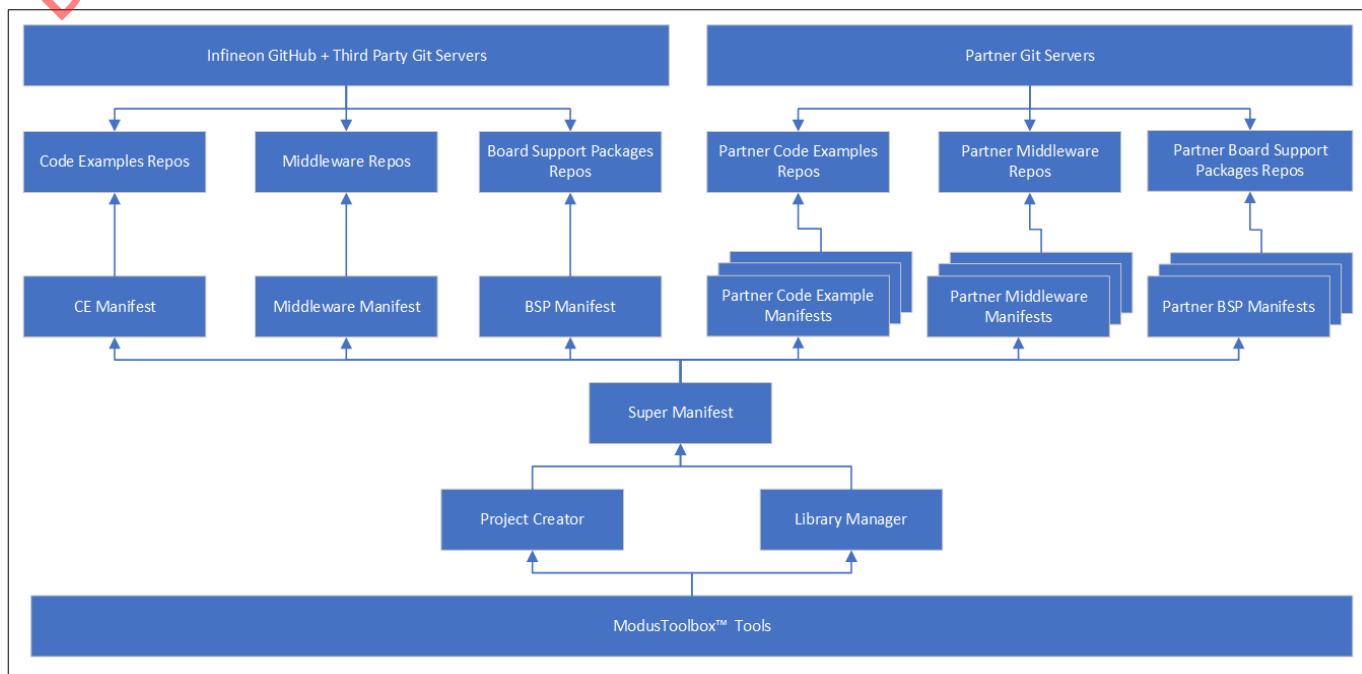
ModusToolbox™ tools (Project Creator and Library Manager) point to a pre-configured super manifest location on GitHub. The super manifest points to the code example, middleware, and BSP manifests as explained earlier. This allows the tools to discover all the software content hosted on GitHub.



~~5 PSoC™ 6 application notes~~

~~DRAFT~~ 5.17.2.5 How does partner integration work?

The partner software content is delivered in a very similar fashion as the content offered by Infineon. As long as the content is hosted on Git servers and conforms to the structure ModusToolbox™ expects, all the content can be brought into ModusToolbox™ tools with the same concepts as shown below:



The super-manifest points to all the partner manifests in addition to Infineon's own manifests so that the partner software content can be easily integrated and discovered in the tool. This allows partners to manage their own content and deployment without any intervention from Infineon once your manifests are included in the super manifest.

~~5 PSoC™ 6 application notes~~

~~DRAFT~~ 5.17.3 Setting up your own Git infrastructure

As we learned in the previous sections, all the software content needs to be hosted on a platform that supports Git-based version control. Other version control systems like Mercurial, Perforce, and Subversion are not supported.

There are several platforms that provide Git hosting solutions today such as GitHub, GitLab, and Bitbucket. ModusToolbox™ can support any hosting platform as long it supports Git version control.

Let's look at the steps to set up your own Git infrastructure.

5.17.3.1 Choosing your Git hosting platform

There is no specific recommendation when it comes to choosing a Git hosting platform. ModusToolbox™ is not optimized to work better with any specific Git hosting platform. The tools only use the underlying Git version control system to bring in the necessary software content.

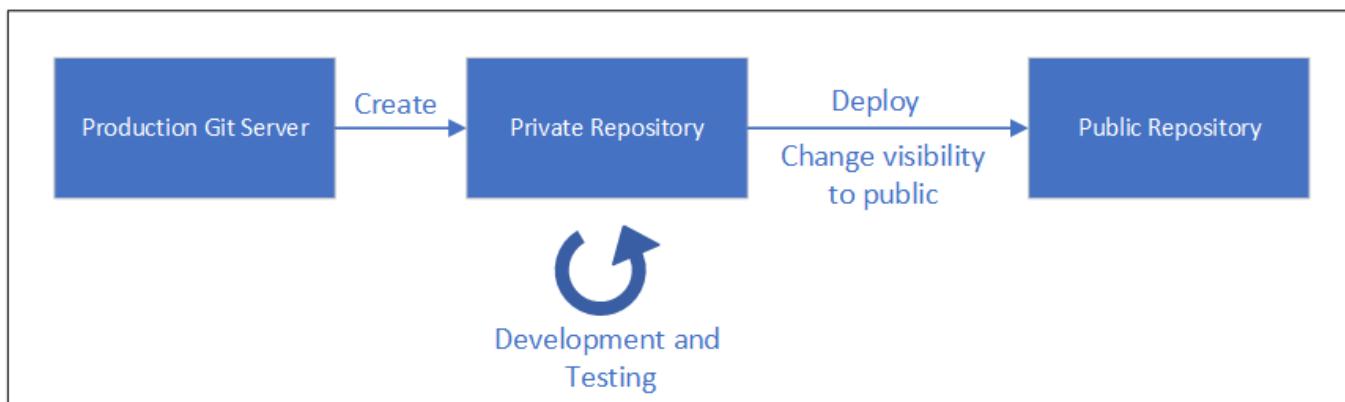
You should make sure that the platform you choose is accessible in all geographic regions in which you wish to supply your content.

5.17.3.2 Choosing your development flow

A development flow is a process used to develop, test, and deploy your software content in a systematic, sequential manner that eliminates errors and meets a certain quality standard. The upcoming sections describe two development workflows that can be used, but assume that you are familiar with Git and Git hosting service platforms to implement such a workflow. How to use Git and Git hosting service platforms are out of scope of this application note. The two most common development flows that can be adopted are as follows:

5.17.3.2.1 Single-stage workflow

A single-stage workflow is when you have a single Git hosting server where the visibility of the repositories is kept private until the development and testing is complete. This is slightly riskier considering that any human error could result in an incorrect or unstable version of the software being released to the public. Certain checks can be put in place to ensure these errors are caught before release.



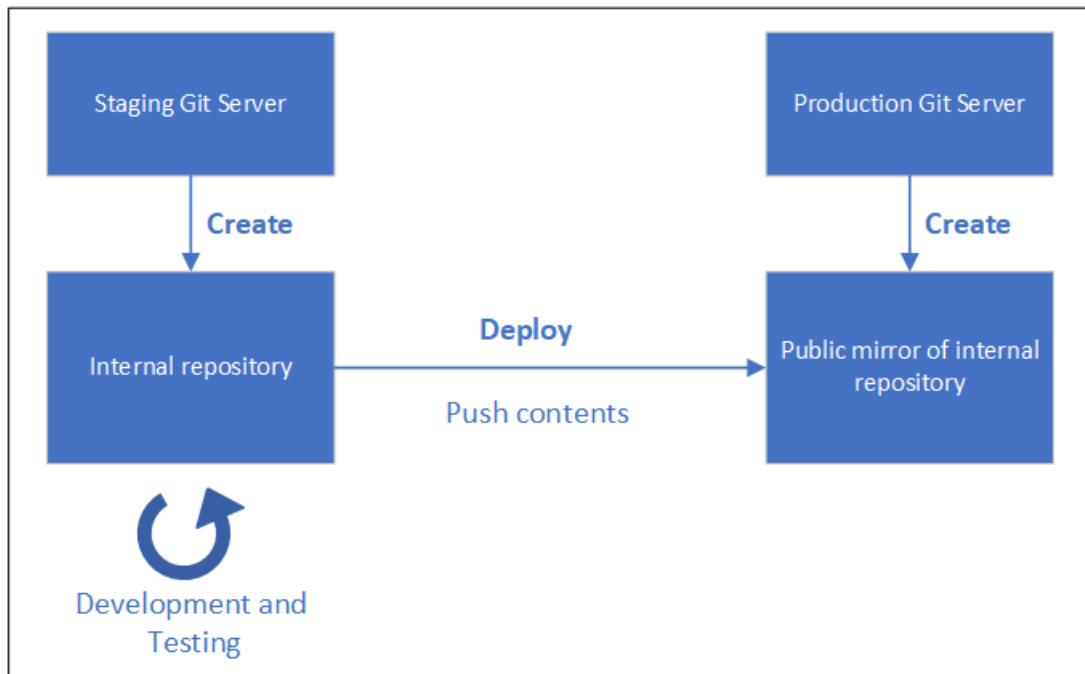
Note: ModusToolbox™ tools does not work directly with private repositories and requires the setup of tokens as explained in the “Access to Private Repositories” section in the [ModusToolbox™ user guide](#) to get access to them. This extra step of setting up the tokens is just a one-time effort.

~~5 PSoC™ 6 application notes~~

~~5.17.3.2.2 Dual-stage workflow~~

A dual-stage setup consists of separate staging and production servers with the following characteristics:

- *Staging* – Internal Git hosting that is private and where the development and testing happens. Any content here is considered functional, reviewed, and pending final approval for release.
- *Production* – External facing Git hosting platform that is public and hosts only fully tested software and released software content.



This setup allows developers to create software content in the staging environment without worrying about any mishaps or errors that could be exposed to the customer because it is internal to the company. Development and testing can be done iteratively in staging. Because the staging environment simulates a near-production-level environment, any issues can be caught before they reach end users. The software is deployed into production only after the content is error-free and completely tested.

The staging environment uses continuous integration and continuous delivery (CI/CD) pipelines to automate the process of testing the software content. The actions are performed on each commit to the repo, or after events like merging a topic branch into the main branch. If GitLab is used for the staging server, the actions performed are determined using the “`.gitlab-ci.yml`” configuration file, which is part of the development repo.

The software content is pushed to production only once all tests have passed. You get to decide how rigorous the testing should be to ensure the quality of the software content offered. See [CI/CD documentation](#) for more information on setting up such automated test pipelines.

Infineon uses the dual-stage development flow for creating its ModusToolbox™ software content.

5 PSoC™ 6 application notes

5.17.3.3 Designing your internal staging setup

The internal staging setup can implement CI/CD pipelines which are a set of automated jobs that perform particular tests based on some rules. These pipelines allow the software to be fully tested whenever there is any change to the repo. It is left to the partners to decide which jobs are necessary in the pipeline and which jobs are optional. Some examples of the types of testing jobs that can be implemented are as follows:

- *Documentation test* – This test validates the documentation. It goes through documentation such as markdown files to check if a particular template is being followed and if all the required sections exist. Additionally, all the links in the document can be tested to find any broken ones.
- *Software test* – This test validates the working of the code against all the constraints such as BSPs, ModusToolbox™ tools versions, toolchains, cross-platform support (Windows, Linux, macOS), and build configurations, and provides information on any errors or warnings that the code may have.
- *Code coverage test* – This test allows the code to be tested for any poor coding practices and conditions that can be optimized.

Format	Build	Test	Coverage
✓ test-ce-structure ! validate-yml-config	✓ build-ce-default-config	✓ test-ce-sanity-linux ✓ test-ce-sanity-macos ✓ test-ce-sanity-windows	! build-ce-github-assets ✓ build-ce-matching-bsps ✓ build-ce-mtb-min-version

5.17.3.4 Designing your external production setup

The external production setup can implement the same CI/CD pipelines (explained in the previous section) to test the software content in a dual-stage setup but it is redundant because the staging environment mirrors the production one and has fully tested the content. However, in a single-stage setup, these pipelines might be necessary to help test the code before the repository goes public.

5 PSoC™ 6 application notes**5.17.4 Creating your own software content**

This section explains the procedure and best practices for going about the creation of software content. It is recommended to follow the steps explained in the upcoming sections so that the content developed works seamlessly with ModusToolbox™. All content should be developed using the latest 2.4 version of ModusToolbox™ for best results but the steps are applicable for all content developed using ModusToolbox™ 2.2 and later.

Note: *This section assumes that you are familiar with creating applications in ModusToolbox™ and now want to create your own content. See the [ModusToolbox™ user guide](#) if this is not the case. Additional training material can be found [here](#).*

5.17.4.1 Creating a code example

All current ModusToolbox™ code examples can be found through the GitHub [code example page](#). There you will find links to examples for the Bluetooth® SDK, PSoC™ 6 MCU, and PSoC™ 4 device among others. You can also create any of the released code examples using the ModusToolbox™ Project Creator tool. Let's look at the steps to create your own code example.

1. Choosing a starter application
2. Choosing the name
3. Choosing the title
4. Adding the source files
5. Adding middleware
6. Adding the End User License Agreement (EULA)
7. Create a Git repository
8. Create a topic branch
9. Testing the code example
10. Merging into mainline
11. Creating the release package

5.17.4.1.1 Choosing a starter application

Based on the application you want to develop and the device you want to target, open Project Creator and choose a BSP and code example that provides a good starting point. You can always use the empty application if you do not know where to begin. Before you click **Create**, choose an appropriate name for the code example by following the instructions provided in section [Choosing the name](#).

5.17.4.1.2 Choosing the name

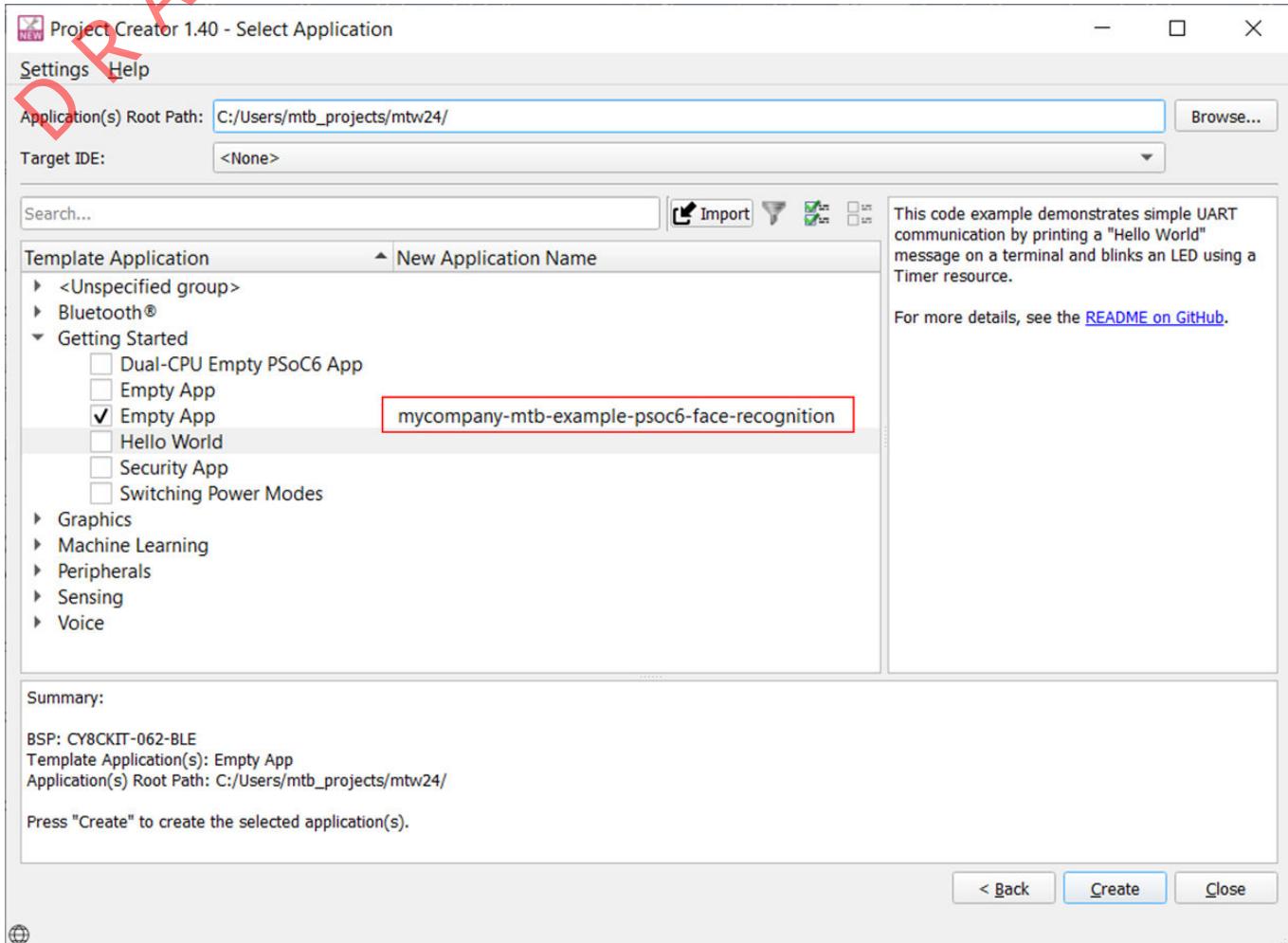
The name of the code example should be unique and should conform to the following best practices:

- Should be short (avoid long file paths)
- All lower-case, with words separated with hyphens ‘-’
- Prefixed with “<partner>-mtb-example-“

Some examples include “mycompany-mtb-example-psoc6-object-detection-using-ml” or “mycompany-mtb-example-psoc6-face-recognition”.

5 PSoC™ 6 application notes

Add this name in the text-box under “New Application Name” as illustrated below and click **Create**.



Once the code example generation is complete, update the Makefile to set the APPNAME make variable to the name of the code example. For example,

```
APPNAME=mycompany-mtb-example-psoc6-face-recognition
```

5 PSoC™ 6 application notes

5.17.4.1.3 Choosing the title

The code example title should be a short descriptive header text and should conform to the following best practices:

- Informs the user of the scope and content of the code example
- It can contain the device family or platform name
- It does not use trademark symbols, superscripts, or subscripts

Some examples include “MyCompany PSoC6 Face Recognition with ML” or “MyCompany PSoC6 Object Detection” or “PSoC6 Face Recognition with ML using MyCompany”.

The `Readme.md` file in the code example should be updated with this title.

5.17.4.1.4 Adding the source files

The Project Creator tool is responsible for creating the fully functional application. During project creation, it looks at the `deps` folder to bring in all the dependent libraries to create the fully functional application. Additionally, based on the IDE of choice, other IDE-specific folders/files may be created.

All ModusToolbox™ code examples will contain the following folder/files:

- `deps` – Contains all the dependencies needed for the code example to run
- `Makefile` – Contains all the configuration variables to control the build flow
- `main.c` – Contains the source code

Add your source files that will interact with Infineon’s device driver libraries or other middleware libraries. Here are some of the best practices when developing your source code:

- Use hardware abstraction layer (HAL) APIs if available for the hardware to allow the code example to be portable across multiple device families.
- Use the pin-aliased names available in the BSPs rather than actual pin numbers.
- Use the `abstraction-rtos` library if possible to make the code modular and reusable by users.
- Update the `Readme.md` file to describe the scope, purpose, working, design, and implementation of the code example. See available code example `Readme.md` files for reference.
- Follow other general coding practices like documentation of functions and meaningful variable naming.

If your source files need to interact with your proprietary software, we will cover how to add your proprietary software as a middleware in the next section.

5.17.4.1.5 Adding middleware

This section assumes that you have already created your middleware by following the steps in the [Creating a middleware library](#) section. The `deps` folder in an application contains a number of `.mtb` files. Each library used in the application is identified by a `.mtb` file. This file contains the URL to a Git repository, a commit tag, and a variable for where to put the library on disk.

For example, a `capsense.mtb` file might contain the following line:

```
http://github.com/cypresssemiconductorco/capsense#latestv2.X##$ASSET_REPO$$/capsense/latest-v2.X
```

The build system implements the `make getlibs` command. This command finds each `.mtb` file, clones the specified repository, checks out the specified commit, and collects all the files into the specified directory. Typically, the `make getlibs` command is invoked transparently when you create an application or use the Library Manager, although you can invoke the command directly from a command-line interface.

5 PSoC™ 6 application notes

Adding your middleware to the code example is fairly simple:

1. Create a .mtb file inside the deps folder with the following value: `http://github.com/<partner_name>/<my_middleware>#<commit>#$$ASSET_REPO$$/<my_middleware>/<commit>`
2. Edit the details highlighted in yellow to specify the path to your middleware on your Git hosting platform and also the corresponding commit or release tag to be used. Save the file and close it.
3. Run Library Manager and click **Update**. This runs `make getlibs` in the background which sees the new .mtb file and clones your middleware into the `mtb_shared` folder. You can also use the command line and run `make getlibs` for the same purpose.
4. Use the make variables in the application Makefile to specify the source files, header files, and paths from the middleware to be included by the build system. This is typically the path to the porting layer (see Scenario #2 in [Middleware](#)) in the middleware that supports Infineon chipsets. See the [ModusToolbox™ user guide](#) for more information on available make targets.

You have successfully added support for your middleware in your code example.

Note: *Make sure to copy these changes to the local copy of the repository and push these changes to the upstream Git server.*

5.17.4.1.6 Adding the End User License Agreement (EULA)

Once the code development is complete, add the EULA provided by Infineon as part of the partner agreement into your source directory. The source directory is the one that contains the application Makefile.

5.17.4.1.7 Create a Git repository

Create a repository with the code example name in your Git hosting platform:

- This will be the staging server for the dual-stage workflow.
- This will be the production server with the repository set to *Private* for the single-stage workflow.

Clone the repository using the `git clone` command. A local copy of this repository will now be available.

5.17.4.1.8 Create a topic branch

It is up to the partner to create a separate topic branch or use the main branch to work on the changes. It is recommended to use a topic branch to prevent any accidental updates to the main branch that might break the code for everyone. Use the `git checkout -b <branch_name>` command to create a topic branch.

As a rule of thumb, copy the project files and add it to your local copy of your Git repository you created previously whenever you make significant progress. Push all these changes to the upstream Git server so that you can always track your work and revert to a previously working configuration if needed. See the [gitignore](#) file for reference to understand what files should be pushed and what should be ignored.

5.17.4.1.9 Testing the code example

The code example must be tested to ensure that there are no issues with the build. It must also be tested on the hardware to verify the functionality. Once both the tests have passed, push the code example to the Git repository for automated test CI/CD pipelines to test it against multiple toolchains, tools versions, build configurations, cross-platform support, etc.

~~DEP~~ 5 PSoC™ 6 application notes

5.17.4.1.10 Merging into mainline

Once the code passes all the test pipelines, merge the topic branch into the main branch. Any merge conflicts that may arise should be resolved. If using a dual-stage setup, deploy the contents into production.

5.17.4.1.11 Creating the release package

Create a [release tag](#) from the main branch with the following naming scheme:

```
release-<major_version>-<minor_version>-<patch_version>
```

For example, the first release version will be `release-v1.0.0`. See [release tags](#) of an existing code example to understand what this looks like.

Here's how to decide the version numbers when creating a release package:

- **major version:** Should be incremented only for code or makefile changes that break backward compatibility. This happens when the code no longer works with previous major versions of the tool or ecosystem. Reset the minor and patch version numbers to '0' when this is incremented.
- **minor version:** Should be incremented for code-related changes like changes to source code, file structure or makefile, that DO NOT break backward compatibility. Reset the patch version number to '0' when this is incremented.
- **patch version:** Should be incremented for document or code comment changes only.

An example of the version history for a code example that underwent different types of changes:

Scenario	Category of change	Version number
New content	New	1.0.0
Documentation fix, no changes to source code	Increment patch number	1.0.1
Cosmetic changes to source code, doesn't affect functionality	Increment patch number	1.0.2
Files / folders restructured	Increment minor number	1.1.0
Source code or Makefile changes that do not break backward compatibility	Increment minor number	1.2.0
Source code or Makefile changes that break backward compatibility	Increment major number	2.0.0

In addition to release tags, latest version tags need to be generated whenever there is a change in the major version number. They follow the naming scheme: `latest-v<major_version>.X`.

The latest version tags are used to point to latest versions of the major version of the content available. This helps users to get the latest version of the content. For example, if there are two release tags for a code example such as `release-v1.0.0` and `release-v1.1.0`, the `latest-v1.X` tag will be generated to point to the latest release of the major version library, i.e., `release-v1.1.0`. See the tags generated [here](#) for reference.

Here's how you can create a latest version tag:

1. Create a [latest version tag](#) from the main branch with the naming scheme specified above and filling in the major version available.
2. Whenever there is an update to the content, the latest version tag must be moved to point to the latest version. For example, during initial deployment of content, `latest-v1.X` will point to the same commit as `release-v1.0.0` tag. Whenever the content is updated to `release-v1.1.0`, the `latest-v1.X` should be updated to point to the same commit as `release-v1.1.0` tag.

~~DEAF~~ 5 PSoC™ 6 application notes

5.17.4.2 ~~DEAF~~ Creating a middleware library

There is no generic way to define what a middleware library should look like. It could be a firmware SDK, a binary or “.a” static library that users can link in their project. In any case, there are certain general guidelines to be followed.

To create a middleware library that is supported by ModusToolbox™, the following should be considered:

- It should follow any of the naming schemes listed below if the middleware library is developed specifically to support ModusToolbox™:
 - <partner_name>-middleware (e.g., mycompany-middleware)
 - <partner_name>-middleware-<feature_name> (e.g., mycompany-middleware-remote-debugging)
 - <partner_name>-middleware-<tool_name> (e.g., mycompany-middleware-graphics-wizard)
 - <partner_name>-middleware-<application> (e.g., mycompany-middleware-image-processing)
- If the middleware library is not developed to support ModusToolbox™ specifically, the naming scheme is left to the discretion of the partners.
- It should be hosted as a Git repository on any Git hosting platform. The name of the Git repository should be the same as the name of the middleware library.
- It should have a valid release and latest tag to prevent applications that use it from breaking because the contents of the middleware might change.
- It should have appropriate documentation to explain the APIs and how to add and use the middleware in an application.
- It can be easily integrated with the make-based flow in ModusToolbox™.
- Constraints should be specified in the documentation such as capabilities of the hardware required, supported operating systems, toolchains, and ModusToolbox™ tools versions.

Once the middleware is developed, push the changes to the upstream Git server and create a release tag similar to the one described in section [Creating the release package](#).

If the middleware has dependencies on other libraries (i.e., it needs other libraries for it to work), then see [Adding middleware dependencies](#) to understand how to specify these dependencies. For example, a partner that offers cloud-based services could require wifi-connection-manager and http-client libraries to establish a connection to the cloud. This can be specified as a dependency to their middleware to make sure the right dependent libraries are brought in whenever the middleware is used.

5.17.4.3 ~~DEAF~~ Creating a BSP

The BSPs have a specific folder structure, files, and Makefiles to allow ModusToolbox™ applications to build correctly. To create your own BSP, ModusToolbox™ supports the make command `make bsp` that creates a custom BSP based on a particular MCU device and optional connectivity module. The steps to create your own custom BSP is provided in the [Creating a Custom BSP User Guide](#).

Once your BSP is created, create a repository on your Git hosting platform with the following best practices:

- Repo name must be prefixed with “TARGET_”
- The BSP name should be all uppercase, with words separated by hyphens “-“.
- Include documentation for the BSP with kit features, contents, default configuration, etc.

For example, “TARGET_MY-KIT-062S2-43012” or “TARGET_MY-SENSOR-SHIELD”.

The BSP contains the `design.modus` file inside the `COMPONENT_BSP_DESIGN_MODUS` folder. Use the Device Configurator to open this file and define all the pin aliases, peripherals, clocks, and any other default settings of the kit.

Once the BSP is developed, push the changes to the upstream Git server and create a release tag similar to the one described in section [Creating the release package](#).

5 PSoC™ 6 application notes~~DRAFT~~
5.17.5 Creating your own manifest

By default, ModusToolbox™ tools look for Infineon's manifest files maintained on Infineon's GitHub server. So, the initial list of BSPs, code examples, and middleware available to use are limited to our manifest files. Once your content is completely integrated into ModusToolbox™, your manifest files will be included in the default Infineon super manifest. However, during development, you can create your own manifest files on your servers or locally on your machine, and you can override or add to where ModusToolbox™ tools look for manifest files. To do that, you first need to create manifest files for your BSPs, code examples, and middleware. You will then create a super manifest that points to these manifest files.

Let's go through the steps to create these manifests in the following sections.

5.17.5.1 Creating repositories

Create repositories on your Git hosting platform for each manifest. Each manifest file should reside in its own individual repository to allow the developer to update, maintain, and revert the manifests over time. That is, you should have one repository for a BSP manifest, one for a code example manifest, and one for a middleware manifest. You only need a repository for the types of content that you will be creating. For example, if you are not creating any BSPs, you don't need a BSP manifest.

The repositories need to use the following naming scheme based on the manifest type:

- Super manifest: <partner_name>-super-manifest
- App manifest: <partner_name>-ce-manifest
- Middleware manifest: <partner_name>-mw-manifest
- BSP manifest: <partner_name>-bsp-manifest

For example, the super manifest repository can be named `mycompany-super-manifest`.

The following repositories can be used for reference on how to create them and understand what goes in them:

- Super manifest: [partner-super-manifest](#)
- App manifest: [partner-ce-manifest](#)
- Middleware manifest: [partner-mw-manifest](#)
- BSP manifest: [partner-bsp-manifest](#)

Note: *ModusToolbox™ tools does not work directly with private repositories and requires the setup of tokens as explained in the “Access to Private Repositories” section in the [ModusToolbox™ user guide](#) to get access to them. This extra step of setting up the tokens is just a one-time effort.*

5 PSoC™ 6 application notes

5.17.5.2 ~~DRAFT~~ Creating your code example manifest

The code example manifest is used to point to URLs of code examples. Inside the partner-ce-manifest repository, create a file named partner-ce-manifest-fv2.xml.

Open the partner-ce-manifest-fv2.xml in an editor of your choice. The code example manifest is essentially an XML with the following base structure:

Code Listing 1

```
<apps version="2.0">
  <app keywords="for keywords">
    <name>code example name</name>
    <category>code example category</category>
    <id>code-example-id-without-spaces</id>
    <uri>code example URL</uri>
    <description>Hi I am a code example</description>
    <req_capabilities>code example capabilities</req_capabilities>
    <versions>
      <version flow_version="2.0" tools_min_version="MTB tools minimum version"
req_capabilities_per_version="bsp_gen3">
        <num>Name of the branch</num>
        <commit>exact branch of the repo to be used</commit>
      </version>
    </versions>
  </app>
</apps>
```

The file starts with `<apps>...</apps>` as the root labels. Each code example is described within an `<app></app>` section. Multiple code examples can thus be added using the `<app>` labels within the root `<apps>` section.

Code Listing 2

```
<apps version="2.0">
  <app keywords="psoc6,partner,demo">
    // body of the code example 1
  </app>
  <app keywords="psoc6">
    // body of the code example 2
  </app>
</apps>
```

~~5 PSoC™ 6 application notes~~

Update the body of the code example using the guidance for the fields described below to add details of your CE:

Field / attribute	Description	Example
app:keywords	<p>List of labels which helps identify the given code example using the Project Creator search feature.</p> <ul style="list-style-type: none"> ✓ Multiple keywords are supported as a comma-delimited list. ✓ Allowed: chars, nums, spaces, hyphen, underscore, period ✓ Best practice: include keywords used in CE title, req_capabilities attribute 	<app keywords="psoc6,led,starter,hello world,mtb-flow">
<name>	<p>Name of the CE as it would show up in the Project Creator.</p> <ul style="list-style-type: none"> ✓ Use a short descriptive text. The CE title is a good starting point. ✗ Avoid including device family or SDK name (example: "PSoC 6") as part of this field. ✗ Do not use special characters like hyphens, colons, underscores, bracket or slashes in this field. Spaces are allowed. 	<name>Hello World</name>
<category>	Value which enables showing CEs in categorized (grouped) list. Check existing categories to determine where the CE fits best.	<category>Voice</category>
<id>	Unique ID of the CE.	<id>mtb-example-psoc6-hello-world</id>
<uri>	Path to the CE's GitHub repo. No trailing or leading spaces.	<uri> https://github.com/Infineon/mtb-example-psoc6-hello-world </uri>
<description>	<p>Short description of the CE. This text shows up in the description area of the Project Creator while selecting the CE.</p> <p>✗ Do not use special characters like ™ or ® or smart (curly) braces.</p> <p>✓ This can be used to include a link to company website and sales.</p>	<description>This code example demonstrates the implementation of simple UART communication and blinks an LED using a Timer resource using PSoC 6 MCU.</description>

5 PSoC™ 6 application notes

Field / attribute	Description	Example
<req_capabilities>	<p>Capabilities required by the CE which must be provided by the board (BSP). CEs will be listed in the Project Creator only for those BSPs which provide these capabilities using the <prov_capabilities> field in the BSP manifest. Use this to ensure that the CE is listed only for supported kits. See Adding BSP capabilities for a list of available capabilities and their descriptions.</p> <p>✓ Enter a space-separated list of capabilities the CE requires. All the required capabilities must be satisfied by a BSP for the CE to show up in Project Creator for that BSP.</p>	psoc6
version:flow_version	<p>Specifies flow version, required/identified by tools > MTB 2.1</p> <p>For the -fv2 manifest, this should be 2.0.</p>	<version flow_version="2.0">
version:num version:commit	<p>These indicate the aliases for the commit hash or branch/tag to fetch. CE should mandatorily contain a release tag i.e., release-v1.0.0 and a latest tag i.e., latest-v1.X. See section Creating the release package for more details.</p> <p>Although the default “main” branch can be used here, it might keep receiving updates and commits might move. It is always better to create a release tag to make the content static.</p>	<pre><versions> <version> <num>Latest v1.X release </num> <commit>latest-v1.X</commit> </version> <version> <num>1.0.0 release</num> <commit>release-v1.0.0</commit> </version> </versions></pre>

5 PSoC™ 6 application notes

Field / attribute	Description	Example
version:tools_min_version version:tools_max_version	<p>[Optional] Specifies min and max ModusToolbox™ tool/patch versions on which the particular <version> of application is displayed.</p> <p>tools_min_version should match the minimum required tools or patch version. For example, 2.2.0 if ModusToolbox™ 2.2 is required without any patches, or 2.2.1 if the 2.2.1 patch is required.</p> <p>All entries in fv2 manifest file MUST contain tools_min_version="2.2.0" or a higher version number.</p> <p>In general, leave tools_max_version blank as examples are expected to work with latest versions of tools.</p> <p><i>Note:</i> <i>The ModusToolbox™ version in the Readme requirements section should match these values.</i></p>	<pre><version tools_min_version="2.2.0" tools_max_version="2.4.0"></pre>
version:req_capabilities_per_version	<p>[Optional] A list of required version-specific capabilities for the given application. This list is treated as an "AND" list in addition to the global req_capabilities.</p> <p>The list is whitespace-delimited. If this attribute is missing or empty, it means that this application has no version-specific capability.</p>	<pre><version req_capabilities_per_version="bsp_gen3 std_crypto"></pre>
app:req_capabilities_v2	[Optional] A list used to restrict the scope of the code example. See Specifying requirements for restricted scope for more information.	req_capabilities_v2="[cy8ckit_062-wifi_bt,cy8ckit_062s2_43012]"

5 PSoC™ 6 application notes

An example of what the manifest file will look with all the details filled in is shown below:

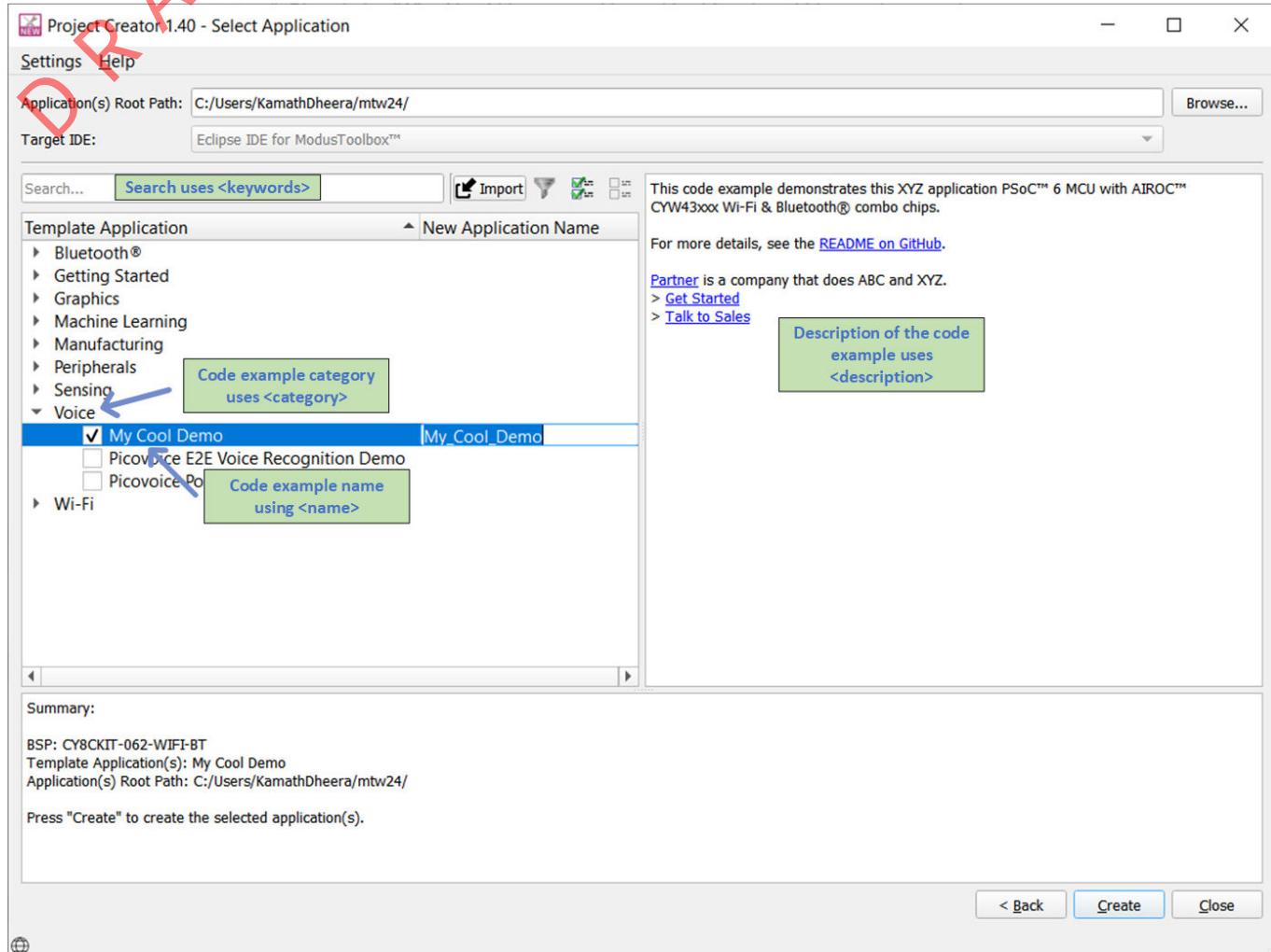
Code Listing 3

```
DRAFT  
<apps version="2.0">  
  <app keywords="psoc6,partner,my,cool,demo">  
    <name>My Cool Demo</name>  
    <category>Voice</category>  
    <id>partner-mtb-example-my-cool-demo</id>  
    <uri>https://github.com/partner/partner-mtb-example-my-cool-demo</uri>  
    <description><![CDATA[This code example demonstrates this XYZ application PSoC™ 6 MCU with  
AIROC™ CYW43xxx Wi-Fi & Bluetooth® combo chips.  
For more details, see the <a href="https://github.com/partner/partner-  
mtb-example-my-cool-demo/blob/master/README.md">README on GitHub</a>. <br><br><a href="https://  
partner-webpage.com/">Partner</a> is a company that does ABC and XYZ.<br> > <a  
href="https://docs.partner-doc-page.com/">Get Started</a><br> > <a href="https://partner-  
webpage.com/contact/">Talk to Sales</a>]]></description>  
    <req_capabilities>psoc6</req_capabilities>  
    <versions>  
      <version flow_version="2.0" tools_min_version="2.4.0"  
req_capabilities_per_version="bsp_gen3">  
        <num>Latest 1.X release</num>  
        <commit>latest-v1.X</commit>  
      </version>  
      <version flow_version="2.0" tools_min_version="2.4.0"  
req_capabilities_per_version="bsp_gen3">  
        <num>1.0.0 release</num>  
        <commit>release-v1.0.0</commit>  
      </version>  
    </versions>  
  </app>  
</apps>
```

Note: *The latest tag is used to bring the latest version of the content available in the repository. This tag must be used to point to the latest version of the library. All the tags mentioned in the manifest must exist in the repository hosting the content. See [Creating the release package](#) to learn more about creating these tags if not already done.*

5 PSoC™ 6 application notes

An illustration of how these fields affect the behavior in project-creator is shown below:



5.17.5.2.1 Adding BSP capabilities

ModusToolbox™ relies on a set of capabilities *provided* by the BSP and *required* by the code examples to determine what is compatible. This ensures that code examples and BSPs can be created and released completely independently of the software tools that present them to customers.

Capabilities	Description
Infineon chips	
psoc4	This board includes a PSoC™ 4 (e.g., PSoC™ 4A-S1, PSoC™ 4A-S2, ...) MCU that can be used to develop firmware on.
psoc6	This board includes a PSoC™ 6 (e.g., PSoC™ 6A-BLE2, PSoC™ 6A-2M, ...) MCU that can be used to develop firmware on.
pmg1	This board includes a EZ-PD™ PMG1 (e.g., CCG3, CCG6, ...) MCU that can be used to develop firmware on.
cat1	This board includes an MCU that is part of CAT1A (or PSoC™ 6) and CAT1B family of devices.
cat2	This board includes an MCU that is part of CAT2 (PSoC™ 4 and PMG1) family of devices.

5 PSoC™ 6 application notes

Capabilities	Description
cat3	This board includes an MCU based on the Infineon XMC™ platform (XMC1xxx, XMC4xxx).
cat4	This board includes an MCU based on the legacy Broadcom 43907 platform (43907, 54907).
xmc	This board includes an XMC™ MCU that can be used to develop firmware on.
xmc1000	This board includes an Arm® Cortex® M0 (CM0) XMC1xxx MCU that can be used to develop firmware on.
xmc4000	This board includes an Arm® Cortex® M4 (CM4) XMC4xxx MCU that can be used to develop firmware on.
xmc7000	This board includes a CM0+ and up to two an Arm® Cortex® M7 (CM7) XMC7xxx MCU that can be used to develop firmware on.
cyw20xxx	This board includes a CYW20xxx (e.g., CYW20735, CYW20820, ...) MCU that can be used to develop firmware on.
cyw20<####>	This board includes the specific connectivity chip (e.g., CYW4343W, CYW43012). This should only be used when the generic 'cyw20xxx' capability is also provided.
cyw43xxx	This board includes a CYW43xxx (e.g., CYW4343W, CYW43012, ...) chip providing Bluetooth® and Wi-Fi functionalities.

Chip capabilities

adc	The MCU on the board contains at least one programmable analog-to-digital converter (ADC) block.
can	The MCU on the board contains at least one Controller Area Network (CAN) block.
comp	The MCU on the board contains at least one analog comparator block.
cyw20xxx_controller	The MCU on the board includes a CYW20xxx (e.g., CYW20819, CYW20706, ...) chip providing Bluetooth® controller functionality (standard HCI, WICED-HCI). A separate MCU is used for the host functionality.
dac	The MCU on the board contains at least one programmable digital-to-analog converter (DAC) block.
dma	The MCU on the board contains at least one programmable DMA block.
flash_<X>k	The MCU on the board contains X kilobytes of internal flash memory, e.g., for a device with 256 KB of flash, the capability would be: flash_256k.
i2c	The MCU on the board contains at least one programmable I2C block.
i2s	The MCU on the board contains at least one programmable I2S block.
lin	The MCU on the board contains at least one programmable Local Interconnect Network (LN) block.
low_power	The MCU and board are capable of running in low-power modes.
lptimer	The MCU on the board contains at least one programmable low-power timer (LPT) block.
mcu_gp	The MCU on the board has a programmable general-purpose MCU (e.g., PSoC™, CYW43907), NOT: 20xxx.
multi_core	The MCU on the board has multiple user-configurable cores (e.g., most PSoC™ 6 MCUs have a CM0+ and a CM4 core)

5 PSoC™ 6 application notes

DRAFT

Capabilities	Description
opamp	The MCU on the board contains at least one programmable operational amplifier block.
qspi	The MCU on the board contains at least one programmable Quad SPI block.
rtc	The MCU on the board contains at least one programmable real-time clock (RTC) block.
secure_boot	The MCU on the board is capable of booting into a secured environment.
smart_io	The board contains provisions to access smart I/O capable pins, even if the pins are multiplexed with other peripherals like CAPSENSE™.
std_crypto	The MCU on the board supports standard cryptography APIs.
uart	The MCU on the board contains at least one programmable UART block.
udb	The MCU on the board has a device that supports configurable Universal Digital Blocks (UDB).
sram_<X>k	The MCU on the board contains X kilobytes of internal SRAM, e.g., for a device with 16 KB of SRAM, the capability would be: sram_16k.
ble	This chip and board are capable of performing Bluetooth® LE communication. DEPRECATED: With the AIROC™ stack unification, going forward the "bt" capability covers all devices with Bluetooth® radios.
bt	This chip and board are capable of performing full Bluetooth® communication.
capsense	This chip supports CAPSENSE™ and the board contains CMOD and/or CINT capacitors to support CAPSENSE™ design. It may not have any widget present though.
capsense_button	The chip supports CAPSENSE™ and the board contains at least one CAPSENSE™ button.
capsense_linear_slider	The chip supports CAPSENSE™ and the board contains at least one CAPSENSE™ linear slider.
capsense_radial_slider	The chip supports CAPSENSE™ and the board contains at least one CAPSENSE™ radial slider.
capsense_touchpad	The chip supports CAPSENSE™ and the board contains at least one CAPSENSE™ touchpad/trackpad.
sdhc	This chip supports SDHC communication and the board contains an SDHC-compatible card port.
usb_device	This chip supports USB device operation and the board contains a USB header.
usb_host	This chip supports USB host operation and the board contains a USB header.
usbpd	This chip supports USB Power Delivery and the board contains a USB header.
wifi	This chip and board are capable of performing full Wi-Fi communication.

Board elements

arduino	This board provides a pinout compatible with Arduino.
enclosure	This board is sealed in an enclosure to showcase liquid tolerance by immersing in liquids.
fram	This board contains at least one memory device that is a F-RAM.

5 PSoC™ 6 application notes

Capabilities	Description
j2	This board contains the an extended J2 pin header beyond the regular Arduino pins (used by some shields). This is only expected to be used if 'arduino' capability is also provided.
led	This board contains at least one user-controllable LED.
memory	This board contains a memory device external to the main processor that can be accessed via SPI, QSPI, ...
nor_flash	This board contains at least one memory device that is a NOR flash.
pot	This board contains an analog potentiometer that can be accessed via a GPIO.
rgb_led	This board contains at least one RGB LED.
serial_led	This board contains at least one RGB LED driven serially using SPI interface.
switch	This board contains at least one user-controllable switch (aka button).
led	The board contains at least one user-controllable LED.

Others

bsp_gen1, bsp_gen2, bsp_gen3, bsp_gen4	<p>Anytime a new major version of a BSP is created, you need to switch the <code>bsp_gen[X]</code> capability to a higher number. This ends up being used by Project Creator in the way the major version of the asset should be used.</p> <p> <code>bsp_gen1</code> - CAT1A v1 <code>bsp_gen2</code> - CAT1A v2, CAT2 v1, CAT3 v1 <code>bsp_gen3</code> - CAT1A v3, CAT2 v2 <code>bsp_gen4</code> - CAT1A v4, CAT1B v1, CAT1C v1, CAT2 v3, CAT3 v2 </p>
---	---

Note: *You don't necessarily have to add all the capabilities. At a basic level just "psoc6" can be used. Other constraints that the code example may have are typically documented in the Readme file.*

A [code example manifest](#) template has been created with details filled out. You can use this directly or modify the fields as required.

5.17.5.2.2 Specifying requirements for restricted scope

The code example manifest lists the capabilities it requires to work using options `<req_capabilities>` or `<req_capabilities_per_version>` described previously. ModusToolbox™ looks at BSPs that provide these capabilities (specified using `<prov_capabilities>` as described in [Adding BSP dependencies](#)) and displays the code examples against only the supported BSPs.

For those special cases when the code example works for specific BSP(s), it is not possible to use `<req_capabilities>` or `<req_capabilities_per_version>` because several BSPs could have the same capabilities. Hence, for a code example with restricted scope where it works only for specific BSPs or devices with a minimum flash size, the manifests provide an additional option `<req_capabilities_v2>` to specify such requirement.

Use `req_capabilities_v2` as per guidance and example provided below:

- `req_capabilities_v2` uses the format "[tag1] [tag2, tag3]" which translates to the app requiring tag1 AND (tag2 OR tag3) Reminder: `req_capabilities` and `req_capabilities_per_version` use the format "tag4 tag5" which translates to the app requiring tag4 AND tag5.
- Overall required capabilities for the app for a particular version is an AND condition of individual lists: `req_capabilities_v2 AND req_capabilities AND req_capabilities_per_version`.

5 PSoC™ 6 application notes

Example 1: For displaying a code example that is supported only by two specific BSPs

Code Listing 4

```
DRAFT  
<apps version="2.0">  
  <app keywords="psoc6,partner,my,cool,demo" req_capabilities_v2="[cy8ckit_062-wifi_bt,cy8ckit_062s2_43012]">  
    <name>My Cool Demo</name>  
    <category>Voice</category>  
    <id>partner-mtb-example-my-cool-demo</id>  
    <uri>https://github.com/partner/partner-mtb-example-my-cool-demo</uri>  
    <description><![CDATA[This code example demonstrates this XYZ application PSoC™ 6 MCU with AIROC™ CYW43xxx Wi-Fi & Bluetooth® combo chips.  
    <br><br>For more details, see the <a href="https://github.com/partner/partner-mtb-example-my-cool-demo/blob/master/README.md">README on GitHub</a>. <br><br><a href="https://partner-webpage.com/">Partner</a> is a company that does ABC and XYZ. <br> > <a href="https://docs.partner-doc-page.com/">Get Started</a><br> > <a href="https://partner-webpage.com/contact/">Talk to Sales</a>]]></description>  
    <req_capabilities>psoc6</req_capabilities>  
    <versions>  
      <version flow_version="2.0" tools_min_version="2.4.0" req_capabilities_per_version="bsp_gen3">  
        <num>Latest 1.X release</num>  
        <commit>latest-v1.X</commit>  
      </version>  
      <version flow_version="2.0" tools_min_version="2.4.0" req_capabilities_per_version="bsp_gen3">  
        <num>1.0.0 release</num>  
        <commit>release-v1.0.0</commit>  
      </version>  
    </versions>  
  </app>  
</apps>
```

5 PSoC™ 6 application notes

Example 2: For displaying a code example that requires minimum flash size

Code Listing 5

```
DRAFT
<apps version="2.0">
  <app keywords="psoc6,partner,my,cool,demo" req_capabilities_v2="[flash_2048k,flash_1024k]">
    <name>My Cool Demo</name>
    <category>Voice</category>
    <id>partner-mtb-example-my-cool-demo</id>
    <uri>https://github.com/partner/partner-mtb-example-my-cool-demo</uri>
    <description><![CDATA[This code example demonstrates this XYZ application PSoC™ 6 MCU with
AIROC™ CYW43xxx Wi-Fi & Bluetooth® combo chips.

<br><br>For more details, see the <a href="https://github.com/partner/partner-
mtb-example-my-cool-demo/blob/master/README.md">README on GitHub</a>. <br><br><a href="https://
partner-webpage.com/">Partner</a> is a company that does ABC and XYZ.<br> > <a
href="https://docs.partner-doc-page.com/">Get Started</a><br> > <a href="https://partner-
webpage.com/contact/">Talk to Sales</a>]]></description>
    <req_capabilities>psoc6</req_capabilities>
    <versions>
      <version flow_version="2.0" tools_min_version="2.4.0">
        <req_capabilities_per_version="bsp_gen3">
          <num>Latest 1.X release</num>
          <commit>latest-v1.X</commit>
        </version>
        <version flow_version="2.0" tools_min_version="2.4.0">
        <req_capabilities_per_version="bsp_gen3">
          <num>1.0.0 release</num>
          <commit>release-v1.0.0</commit>
        </version>
      </versions>
    </app>
  </apps>
```

5 PSoC™ 6 application notes

5.17.5.3 ~~DRAFT~~ Creating your middleware manifest

The middleware manifest is used to point to URLs of middleware. Inside the partner-mw-manifest repository, create a file named `partner-mw-manifest-fv2.xml`.

The middleware manifest file has the following base structure:

Code Listing 6

```
<middleware>
  <middleware>
    <name>Middleware Name</name>
    <id>middleware-id-without-spaces</id>
    <uri>Middleware URL</uri>
    <desc>Middleware Description</desc>
    <category>Middleware</category>
    <req_capabilities>capabilities required for the middleware to work</req_capabilities>
    <versions>
      <version flow_version="1.0,2.0" tools_min_version="2.2.0">
        <num>Version number</num>
        <commit>release tag of middleware</commit>
        <desc>description for release tag</desc>
      </version>
    </versions>
  </middleware>
</middleware>
```

The root of the XML contains the `<middleware> ... </middleware>` section. Details of the middleware are written within another `<middleware>` section within the root `<middleware>` node as shown below:

Code Listing 7

```
<middleware>
  <middleware>
    // body of middleware 1
  </middleware>
  <middleware>
    // body of middleware 2
  </middleware>
</middleware>
```

5 PSoC™ 6 application notes

Update the body of the middleware using the guidance for the fields described below to add details of your middleware:

Element	Description	Example
<name>	A user-friendly name for the middleware. This is what is displayed in the UI. Typically, the name is the same as the GitHub repository name.	partner-middleware
<id>	A unique identifier for the middleware. The manifest processing code will give an error if multiple middleware items have the same ID. Typically, the name is the same as the GitHub repository name.	partner-middleware
<uri>	The URI for the Git repository holding the middleware.	https://github.com/partner/partner-middleware
<desc>	<p>A user-friendly text description of the middleware item. This is meant to be displayed in the UI. Typically, this is one or two sentences that match the GitHub repository "About" text or README.md first paragraph.</p> <p>Note: Some asset descriptions apply advanced formatting like bold/inline text or line separators. In such situations, enclose the text into CDATA section.</p>	<![CDATA[Block device drivers for use with littlefs filesystem. License Disclaimer: Adding this library will also download and add littlefs to your project. It is your responsibility to understand and accept the littlefs license.]]>
<category>	A user-friendly text string that specifies the category for displaying this middleware item in a GUI. It is expected that all middleware in the same category will be shown together in the library management GUI.	Middleware
<req_capabilities>	<p>A list of capabilities that this middleware requires. This list is treated as an "AND" list. That is, all capabilities must be met. The list is whitespace-delimited; each item in the list must be a valid C identifier. If this element is missing or empty, it means that this middleware has no capability requirements. That is, it works with all boards.</p> <p>For the complete list of the provided capabilities that can be "required" by the middleware, see Adding BSP capabilities.</p>	psoc6

5 PSoC™ 6 application notes

All libraries and middleware are shown in the Library Manager tool. An illustration of how these fields affect the behavior in Library Manager is shown below:

Name	Shared	Version
threadx		1.0.0 release
anycloud-ota		5.0.0 release
aws-iot-device-sdk-embedded-C	✓	202103.00
aws-iot-device-sdk-port	✓	2.2.2 release
azure-c-sdk-port		1.2.1 release
azure-sdk-for-c		1.1.0 release
ble-mesh		3.1.0 release
capsense	✓	3.0.0 release
command-console		4.0.0 release
csdadc		2.0.0 release
csdadic		2.10.0 release
dfu		4.20.0 release
emeeprom		2.20.0 release
emfile		1.0.0 release
emwin		6.24.0 release
enterprise-security		2.1.0 release
http-client	✓	1.2.1 release
http-server		2.2.1 release
littlefs		2.4.0 Release
ipa		3.2.0 release
lwip-freeRTOS-integration		1.0.0 release
lwip-network-interface-integration		1.0.0 release
MCUboot	✓	v1.8.1 Cypress
Partner Middleware SDK	✓	1.0.0 release
mqtt		0.31.0
mtb-littlefs		v1.3.0 Release
netxduo-network-interface-integration		v1.0.0 Release
*Partner Middleware SDK	✓	1.0.0 release
picovoice-lib-psoc6		3.4.2 release
porcupine-lib-psoc6		1.0.0 release
secure-sockets	✓	1.0.0 release
smartcoex		2.10.0 release
usbdev		2.1.0 release
wifi-connection-manager	✓	2.3.0 release

Partner Middleware SDK: add <https://github.com/partner/partner-middleware-sdk> 1.0.0 release shared
INFO - Warning: Multiple versions of "core-make" requested. Keeping version "latest-v1.X" and discarding version "release-v1.9.0".

5 PSoC™ 6 application notes

An example of what the middleware manifest file will look like with all the details filled is shown below:

Code Listing 8

```
DRAFT  
<middleware>  
  <middleware>  
    <name>partner-middleware-sdk</name>  
    <id>partner-middleware-sdk</id>  
    <uri>https://github.com/partner/partner-middleware-sdk</uri>  
    <desc>This SDK is used for ABC and XYZ on PSoC6 devices.</desc>  
    <category>Middleware</category>  
    <req_capabilities>cat1</req_capabilities>  
    <versions>  
      <version flow_version="1.0,2.0">  
        <num>Latest v1.X release</num>  
        <commit>latest-v1.X</commit>  
        <desc>latest-v1.X</desc>  
      </version>  
      <version flow_version="1.0,2.0">  
        <num>1.0.0 release</num>  
        <commit>release-v1.0.0</commit>  
        <desc>release-v1.0.0</desc>  
      </version>  
    </versions>  
  </middleware>  
</middleware>
```

Note: *The latest tag is used to bring the latest version of the content available in the repository. This tag must be used to point to the latest version of the library. All tags mentioned in the manifest must exist in the repository hosting the content. See [Creating the release package](#) to learn more about creating these tags if not already done.*

~~5 PSoC™ 6 application notes~~

~~5.17.5.3.1 Adding middleware dependencies~~

Middleware libraries can work standalone or require other middleware or other high-level libraries to work correctly. A middleware dependencies manifest is used to specify the dependencies a middleware may have on other middleware or high-level libraries.

Note: *The dependency manifest defines only dependencies between high-level middleware libraries. There is no need to define the dependencies on the low-level base libraries like HAL, PDL, or Core-Lib. Instead, the BSP dependencies manifest will define such dependencies separately for each BSP.*

For example, an MQTT middleware might have dependencies on Wi-Fi and HTTP libraries to function correctly. ModusToolbox™ offers a number of middleware libraries that can be added dependencies. These middleware libraries are divided into three categories:

- General middleware libraries ([mtb-mw-manifest](#))
- Bluetooth® middleware libraries ([mtb-bt-mw-manifest](#))
- Wi-Fi middleware libraries ([mtb-wifi-mw-manifest](#))

All the above links point to the manifest repositories used by ModusToolbox™ for organizing the middleware libraries. If your library has a dependency on a Wi-Fi library, use the mtb-wifi-mw-manifest repository to find the information required for your dependency manifest. Similarly, if your middleware has a dependency on a general library, use the mtb-mw-manifest repository to find the information required for your dependency manifest.

In each of these repositories, you will find manifest files that have the same typical structure:

Code Listing 9

```
<middleware>
  <name>bmm150</name>
  <id>bmm150</id>
  <uri>https://github.com/BoschSensortec/BMM150-Sensor-API</uri>
  <desc>Bosch BMM-150 Sensor Driver.</desc>
  <category>Peripheral</category>
  <req_capabilities>mcu_gp</req_capabilities>
  <versions>
    <version flow_version="1.0,2.0">
      <num>2.0.0 release</num>
      <commit>bmm150_v2.0.0</commit>
      <desc>2.0.0 release</desc>
    </version>
  </versions>
</middleware>
```

Based on which middleware library should be added as a dependency, locate the name of the middleware using the `<name>` label in the manifest file. For example, if your middleware has a dependency on the BMM150 sensor library, because this is a non-Bluetooth® or a non-Wi-Fi library, you will use the mtb-mw-manifest repository to search for it.

Once you have found the library, note the ID of the middleware in the `<id>` label. This will be used to specify the dependency.

5 PSoC™ 6 application notes

Create a file in the partner-mw-manifest repository named partner-mw-dependencies-manifest.xml. The structure of the middleware dependencies manifest file is as follows:

Code Listing 10

```
<dependencies>
  <depender>
    <id>my-middleware-id</id>
    <versions>
      <version>
        <commit>2.0.0</commit>
        <dependees>
          <dependee>
            <id>dependent-library1-id</id>
            <commit>1.1.0</commit>
          </dependee>
          <dependee>
            <id>dependent-library2-id</id>
            <commit>1.2.0</commit>
          </dependee>
        </dependees>
      </version>
      <version>
        <commit>1.0.0</commit>
        <dependees>
          <dependee>
            <id>dependent-library1-id</id>
            <commit>1.0.0</commit>
          </dependee>
          <dependee>
            <id>dependent-library2-id</id>
            <commit>1.1.0</commit>
          </dependee>
        </dependees>
      </version>
    </versions>
  </depender>
</dependencies>
```

All the middleware dependencies are specified within the root `<dependencies>...</dependencies>` section. The dependencies of each middleware library are specified within the `<depender>` section. Because each middleware library can have dependencies on different versions of other libraries for a particular release, the `<version>` section is used to differentiate the dependencies across versions.

5 PSoC™ 6 application notes

Update the body of the middleware dependencies file using the guidance for the fields described below to add details of your middleware:

Element	Description	Example
<id>	A unique identifier for the middleware. The manifest processing code will give an error if multiple middleware items have the same ID. Typically, the name is the same as the GitHub repository name.	<i>partner-middleware</i>
<commit>	Specifies the tag/branch used for fetching the library	1.0.0
<dependee><id>	Unique identifier of the dependent library	bmm150
<dependee><commit>	Specifies the tag/branch used for fetching the dependent library for a specific version of the depender library.	bmm150_v2.0.0

5 PSoC™ 6 application notes

For example, if your middleware has a dependency on wifi-connection-manager and http-client to function correctly, the completed middleware dependencies file should look like this:

Code Listing 11

```
<dependencies>
  <depender>
    <id>my-middleware-id</id>
    <versions>
      <version>
        <commit>2.0.0</commit>
        <dependees>
          <dependee>
            <id>dependent-library1-id</id>
            <commit>1.1.0</commit>
          </dependee>
          <dependee>
            <id>dependent-library2-id</id>
            <commit>1.2.0</commit>
          </dependee>
        </dependees>
      </version>
      <version>
        <commit>1.0.0</commit>
        <dependees>
          <dependee>
            <id>dependent-library1-id</id>
            <commit>1.0.0</commit>
          </dependee>
          <dependee>
            <id>dependent-library2-id</id>
            <commit>1.1.0</commit>
          </dependee>
        </dependees>
      </version>
    </versions>
  </depender>
</dependencies>
```

The middleware manifest dependencies file reference can be found [here](#).

5.17.5.4 Creating your BSP manifest

The BSP manifest is used to point to URLs of BSPs. Inside the partner-bsp-manifest repository, create a file named partner-bsp-manifest-fv2.xml.

~~5 PSoC™ 6 application notes~~

The BSP manifest file has the following base structure:

~~Code Listing 12~~

```

<boards>
  <board default_location="local">
    <id>CUSTOM-BSP-NAME</id>
    <category>BSP Category</category>
    <board_uri>URL of the BSP</board_uri>
    <chips>
      <mcu>Name of the MCU chip used</mcu>
      <radio>Name of optional radio chip used</radio>
    </chips>
    <name>Name of the kit</name>
    <summary>Summary of the BSP</summary>
    <prov_capabilities>Capabilities of the BSP </prov_capabilities>
    <description> Detailed description of the BSP </description>
    <documentation_url>URL to BSP documentation </documentation_url>
    <versions>
      <version tools_min_version="2.4.0" flow_version="1.0,2.0"
prov_capabilities_per_version="bsp_gen3">
        <num>Latest 0.X release</num>
        <commit>latest-v0.X</commit>
      </version>
      <version tools_min_version="2.4.0" flow_version="1.0,2.0"
prov_capabilities_per_version="bsp_gen3">
        <num>0.5.0 release</num>
        <commit>release-v0.5.0</commit>
      </version>
    </versions>
  </board>
</boards>
```

The root of the XML contains the `<boards> ... </boards>` section. Details of the BSPs are written inside the `<board>` section within the root `<boards>` node as shown below:

~~Code Listing 13~~

```

<boards>
  <board>
    // body of BSP 1
  </board>
  <board>
    // body of BSP 2
  </board>
</boards>
```

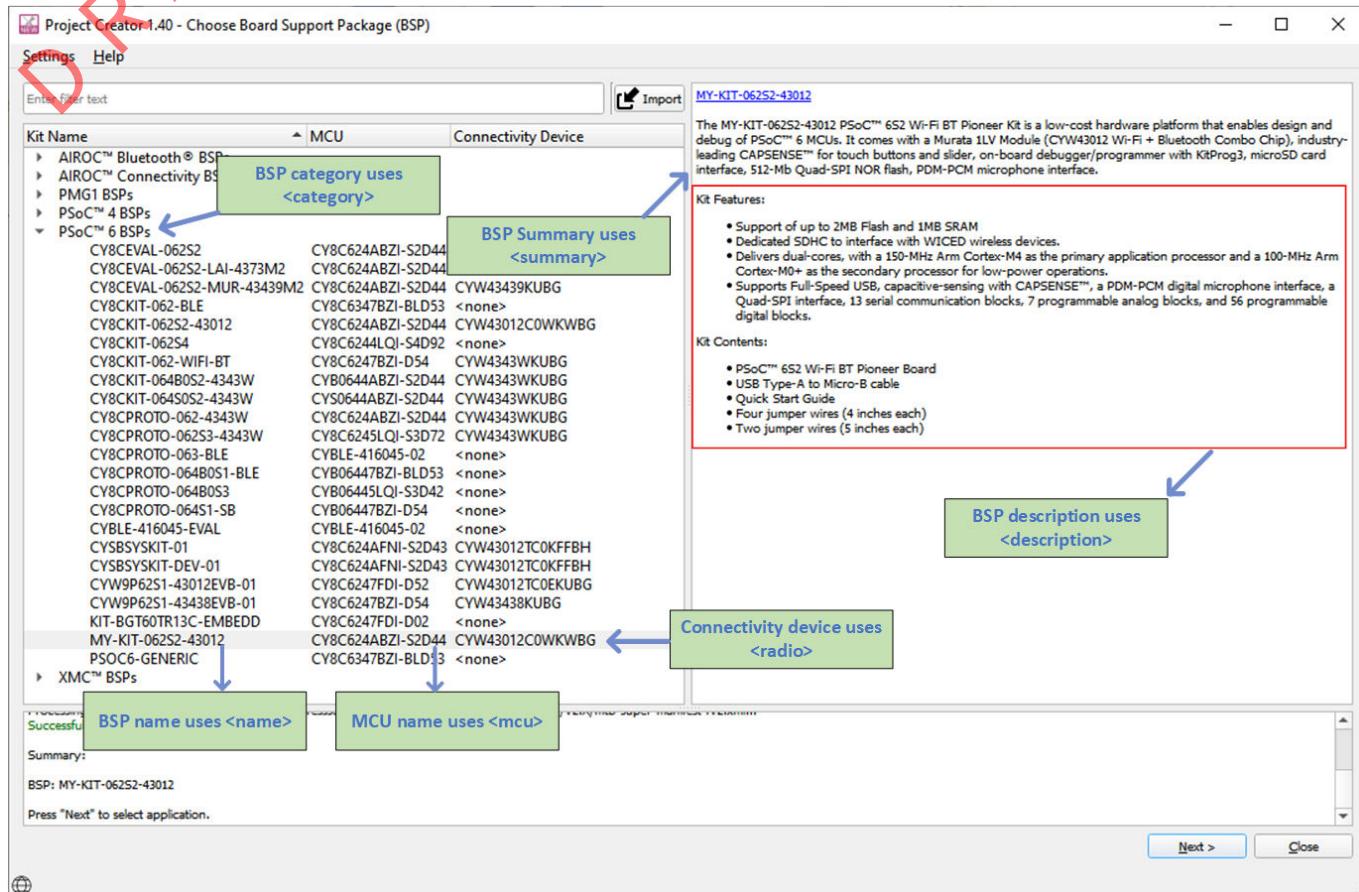
5 PSoC™ 6 application notes

Update the body of the board section using the guidance for the fields described below to add details of your middleware:

Element	Description	Example
board:default_location (optional)	Describes whether the BSP should be placed in the shared or local application folder	<board default_location="local"> or <board default_location="shared">
<id>	A unique identifier for the custom BSP. The manifest processing code will give an error if multiple BSP items have the same ID. Typically, the name is the same as the GitHub repository name.	MY-CUSTOM-BSP
<board_uri>	The URI for the Git repository holding the custom BSP	https://github.com/partner/TARGET_MY-CUSTOM-BSP
<description>	A user-friendly text description of the BSP item. This is meant to be displayed in the UI. Typically, this is a few sentences that match the GitHub repository "About" text or the first paragraph in README.md. Note: <i>Some asset descriptions apply advanced formatting like bold/inline text or line separators. In such situations, enclose the text into CDATA section.</i>	<![CDATA[My Custom BSP is an evaluation kit with the following features.]]>
<category>	A user-friendly text string that specifies the category for displaying this BSP item in a GUI. It is expected that all BSP in the same category will be shown together in the Project Creator GUI.	PSoC® 6 BSPs. Note that ® is the special code representation of the trademark symbol.
<prov_capabilities>	A list of capabilities that this BSP provides. This list is treated as an "AND" list. That is, all the capabilities that are provided. The list is whitespace-delimited; each item in the list must be a valid C identifier. If this element is missing or empty, it means that this BSP has no capability. For the complete list of the provided capabilities by the BSP, see Adding BSP capabilities .	psoc6

5 PSoC™ 6 application notes

All BSPs are shown in the Project Creator tool. An illustration of how these fields affect the behavior in Project Creator is shown below:



An example of what the BSP manifest file will look like with all the details filled in is shown below:

5 PSoC™ 6 application notes

Code Listing 14

DRAFT

```
<board default_location="local">
  <id>MY-KIT-062S2-43012</id>
  <category>PSoC® 6 BSPs</category>
  <board_uri>https://github.com/partner/MY-KIT-062S2-43012</board_uri>
  <chips>
    <mcu>CY8C624ABZI-S2D44</mcu>
  </chips>
  <name>MY-KIT-062S2-43012</name>
  <summary>The BSP MY-KIT-062S2-43012 is the next generation board for XYZ applications</summary>
  <prov_capabilities>adc arduino capsense capsense_button capsense_linear_slider cat1 comp
dma flash_2048k hal i2c i2s j2 led low_power lptimer mcu_gp memory memory_qspi multi_core
nor_flash pdm psoc6 qspi rgb_led rtc sdhc smart_io spi sram_1024k std_crypto switch uart</prov_capabilities>
  <description><![CDATA[
    <div class="category">Kit Features:</div><ul>
      <li>Ready-to-Use CAPSENSE® Trackpad</li>
      <li>EZ-BLE PRoC module</li>
      <li>Potentiometer</li>
      <li>Rechargeable coin-cell battery</li>
    </ul><p>
    <div class="category">Kit Contents:</div><ul>
      <li>MY-KIT-062S2-43012</li>
      <li>USB Standard-A to Micro-B cable</li>
      <li>ABC Sensor</li>
      <li>Four press-fit connectors (for Arduino headers)</li>
      <li>Four jumper wires</li>
      <li>Quick Start Guide</li>
    </ul>
  ]]></description>
  <documentation_url>https://www.partner.com/documentation/development-kitsboards/MY-
KIT-062S2-43012</documentation_url>
  <versions>
    <version tools_min_version="2.2.0" flow_version="1.0,2.0"
prov_capabilities_per_version="bsp_gen2">
      <num>Latest v1.X release</num>
      <commit>latest-v1.X</commit>
    </version>
    <version tools_min_version="2.2.0" flow_version="1.0,2.0"
prov_capabilities_per_version="bsp_gen2">
      <num>1.1.0 release</num>
      <commit>release-v1.1.0</commit>
    </version>
    <version tools_min_version="2.2.0" flow_version="1.0,2.0"
prov_capabilities_per_version="bsp_gen2">
      <num>1.0.0 release</num>
      <commit>release-v1.0.0</commit>
    </version>
  </versions>
</board>
```

5 PSoC™ 6 application notes

```
</versions>
</board>
```

The BSP manifest file reference can be found [here](#).

5.17.5.4.1 Adding BSP dependencies

All BSPs rely on low-level device-specific libraries to work correctly. A BSP dependencies manifest is used in to specify the dependencies a BSP has on the lower-level libraries.

For example, a PSoC™ 6 BSP has dependencies on PSoC™ 6 Peripheral Driver Library (mtb-pdl-cat1), PSoC™ 6 Hardware Abstraction Layer library (mtb-hal-cat1), core libraries (core-lib), etc. ModusToolbox™ offers a number of libraries that can be added as a dependency. These libraries are divided into three categories:

- General middleware libraries ([mtb-mw-manifest](#))
- Bluetooth® middleware libraries ([mtb-bt-mw-manifest](#))
- Wi-Fi middleware libraries ([mtb-wifi-mw-manifest](#))

All the above links point to the manifest repositories used by ModusToolbox™ for organizing the middleware libraries. If your library has a dependency on a Wi-Fi library, use the mtb-wifi-mw-manifest repository to find the information required for your dependency manifest. Similarly, if your middleware has a dependency on a general library, use the mtb-mw-manifest repository to find the information required for your dependency manifest.

In each of these repositories, you will find manifest files that have the same typical structure:

Code Listing 15

```
<middleware>
  <name>mtb-pdl-cat1</name>
  <id>mtb-pdl-cat1</id>
  <uri>https://github.com/cypresssemiconductorco/mtb-pdl-cat1</uri>
  <desc>The Peripheral Driver Library (PDL) integrates device header files, startup code, and low-level peripheral drivers into a single package.</desc>
  <category>Core</category>
  <req_capabilities>cat1</req_capabilities>
  <versions>
    <version flow_version="1.0,2.0" tools_min_version="2.2.0">
      <num>2.4.0 release</num>
      <commit>release-v2.4.0</commit>
      <desc>2.4.0 release</desc>
    </version>
    <version flow_version="1.0,2.0" tools_min_version="2.2.0">
      <num>2.3.1 release</num>
      <commit>release-v2.3.1</commit>
      <desc>2.3.1 release</desc>
    </version>
    </version>
  </versions>
</middleware>
```

Based on the middleware library that should be added as a dependency, locate the name of the middleware using the `<name>` label in the manifest file. For example, if your BSP has a dependency on the PSoC™ 6 PDL,

5 PSoC™ 6 application notes

because this is a non-Bluetooth® or a non-Wi-Fi library, you will use the mtb-mw-manifest repository to search for it.

Once you have found the library, note the ID of the middleware in the `<id>` label. This will be used to specify the dependency.

Create a file in the partner-bsp-manifest repository named `partner-bsp-dependencies-manifest.xml`. The structure of the BSP dependencies manifest file is as follows:

Code Listing 16

```
<dependencies version="2.0">
  <depender>
    <id>my-bsp-id</id>
    <versions>
      <version>
        <commit>2.0.0</commit>
        <dependees>
          <dependee>
            <id>dependent-library1-id</id>
            <commit>1.1.0</commit>
          </dependee>
          <dependee>
            <id>dependent-library2-id</id>
            <commit>1.2.0</commit>
          </dependee>
        </dependees>
      </version>
      <version>
        <commit>1.0.0</commit>
        <dependees>
          <dependee>
            <id>dependent-library1-id</id>
            <commit>1.0.0</commit>
          </dependee>
          <dependee>
            <id>dependent-library2-id</id>
            <commit>1.1.0</commit>
          </dependee>
        </dependees>
      </version>
    </versions>
  </depender>
</dependencies>
```

All BSP dependencies are specified within the root `<dependencies>...</dependencies>` section. The dependencies of each BSP library are specified within the `<depender>` section. Because each BSP library can have dependencies on different versions of other libraries for a particular release, the `<version>` section is used to differentiate the dependencies across versions.

5 PSoC™ 6 application notes

Update the body of the BSP dependencies file using the guidance for the fields described below to add details of your middleware:

Element	Description	Example
<id>	A unique identifier for the BSP. The manifest processing code will give an error if multiple middleware items have the same ID. Typically, the name is the same as the GitHub repository name.	PARTNER-MY-KIT-062S2-43012
<commit>	Specifies the tag/branch used for fetching the library	1.0.0
<dependee><id>	Unique identifier of the dependent library	mtb-pdl-cat1
<dependee><commit>	Specifies the tag/branch used for fetching the dependent library for a specific version of the dependee library	release_v2.0.0

5 PSoC™ 6 application notes

For example, if your BSP has a dependency on PSoC™ 6 PDL and PSoC™ 6 HAL to function correctly, the completed BSP dependencies file should look like this:

Code Listing 17

```
<dependencies>
  <depender>
    <id>PARTNER-MY-KIT-062S2-43012</id>
    <versions>
      <version>
        <commit>latest-v1.X</commit>
        <dependees>
          <dependee>
            <id>mtb-pdl-cat1</id>
            <commit>latest-v2.X</commit>
          </dependee>
          <dependee>
            <id>mtb-hal-cat1</id>
            <commit>latest-v1.X</commit>
          </dependee>
        </dependees>
      </version>
      <version>
        <commit>release-v1.0.0</commit>
        <dependees>
          <dependee>
            <id>mtb-pdl-cat1</id>
            <commit>latest-v2.X</commit>
          </dependee>
          <dependee>
            <id>mtb-hal-cat1</id>
            <commit>latest-v1.X</commit>
          </dependee>
        </dependees>
      </version>
    </versions>
  </depender>
</dependencies>
```

The BSP manifest dependencies file reference can be found [here](#).

5 PSoC™ 6 application notes

5.17.5.5 Creating your super manifest

The super manifest is used to point to the URLs of your code example, middleware, and BSP manifest files. Inside the partner-super-manifest repository, create a file named partner-super-manifest-fv2.xml.

The super manifest has the following base format:

Code Listing 18

```
<super-manifest version="2.0">
  <board-manifest-list>
    <board-manifest>
      <uri>https://github.com/partner/partner-bsp-manifest/raw/main/partner-bsp-manifest.xml</uri>
    </board-manifest>
  </board-manifest-list>
  <app-manifest-list>
    <app-manifest>
      <uri>https://github.com/partner/partner-ce-manifest/raw/main/partner-ce-manifest-fv2.xml</uri>
    </app-manifest>
  </app-manifest-list>
  <middleware-manifest-list>
    <middleware-manifest>
      <uri>https://github.com/partner/partner-mw-manifest/raw/main/partner-mw-manifest.xml</uri>
    </middleware-manifest>
  </middleware-manifest-list>
</super-manifest>
```

At the root, the `<super-manifest>` label is used to identify that is a super manifest type and flow version of the manifest is v2.0. It then uses `<board-manifest-list>`, `<app-manifest-list>`, and `<middleware-manifest-list>` sections for listing the BSPs, code examples, and middleware libraries to be brought in.

The URL to the manifest XMLs must be in raw format. To create this link, use the following scheme:

```
<link to repo>/raw/<branch_name>/<manifest_name>.xml
```

For example, a repository named [partner-ce-manifest](#) and having the manifest file [partner-ce-manifest-fv2.xml](#) on branch “master” will have the following link:

<https://github.com/Infineon/partner-ce-manifest/raw/master/partner-ce-manifest-fv2.xml>

5 PSoC™ 6 application notes

To specify the BSP manifests, each BSP manifest URIs can be added using the `<board-manifest>` sections within the `<board-manifest-list>` using the `<uri>` field. For example, if you have two different BSP manifests you want to point to, it can be done this way:

Code Listing 19

```
<board-manifest-list>
  <board-manifest>
    <uri>https://github/partner/partner-bsp-manifest/raw/main/partner-bsp-manifest.xml</uri>
  </board-manifest>
  <board-manifest>
    <uri>https://github/partner/partner-bsp-manifest/raw/main/other-bsp-manifest.xml</uri>
  </board-manifest>
</board-manifest-list>
```

Similarly, to specify the code example manifests, each of the code example manifest URIs can be added using the `<app-manifest>` sections within the `<app-manifest-list>` using the `<uri>` field. For example, if you have two different code example manifests that you want to point to, it can be done this way:

Code Listing 20

```
<app-manifest-list>
  <app-manifest>
    <uri>https://github.com/partner/partner-ce-manifest/raw/main/partner-ce-manifest-fv2.xml</uri>
  </app-manifest>
  <app-manifest>
    <uri>https://github.com/partner/partner-ce-manifest/raw/main/other-ce-manifest-fv2.xml</uri>
  </app-manifest>
</app-manifest-list>
```

Similarly, to specify the middleware manifests, each middleware manifest URI can be added using the `<middleware-manifest>` sections within the `<middleware-manifest-list>` using the `<uri>` field. For example, if you have two different code example manifests that you want to point to, it can be done this way:

Code Listing 21

```
<middleware-manifest-list>
  <middleware-manifest>
    <uri>https://github.com/partner/partner-mw-manifest/raw/main/partner-mw-manifest.xml</uri>
  </middleware-manifest>
  <middleware-manifest>
    <uri>https://github.com/partner/partner-mw-manifest/raw/main/other-mw-manifest.xml</uri>
  </middleware-manifest>
</middleware-manifest-list>
```

5 PSoC™ 6 application notes

The super manifest should finally look like this once all the details have been filled in:

Code Listing 22

```
<super-manifest version="2.0">
  <board-manifest-list>
    <board-manifest>
      <uri>https://github.com/partner/partner-bsp-manifest/raw/main/partner-bsp-manifest.xml</uri>
    </board-manifest>
  </board-manifest-list>
  <app-manifest-list>
    <app-manifest>
      <uri>https://github.com/partner/partner-ce-manifest/raw/main/partner-ce-manifest-fv2.xml</uri>
    </app-manifest>
  </app-manifest-list>
  <middleware-manifest-list>
    <middleware-manifest>
      <uri>https://github.com/partner/partner-mw-manifest/raw/main/partner-mw-manifest.xml</uri>
    </middleware-manifest>
  </middleware-manifest-list>
</super-manifest>
```

The super manifest file reference can be found [here](#).

Note: You will probably only have a single manifest each for BSP, code examples and middleware unless you need to bifurcate the manifests for organizing content a certain way.

Note: In cases when you don't deal with a particular manifest, that entire list can be omitted from the super manifest. For example, if you don't have a BSP manifest, the <board-manifest-list> .. </board-manifest-list> section can be removed entirely.

5 PSoC™ 6 application notes

5.17.5.6 Specifying the dependency manifests

If the middleware or BSP manifests have a dependency manifest attached to them, they should be specified in the super manifest using the dependency-url field in the following manner:

Code Listing 23

```
<super-manifest version="2.0">
  <board-manifest-list>
    <board-manifest dependency-url="https://github.com/Infineon/partner-bsp-manifest/raw/master/
    partner-bsp-dependencies-manifest.xml">
      <uri>https://github.com/Infineon/partner-bsp-manifest/raw/master/partner-bsp-manifest-
    fv2.xml</uri>
    </board-manifest>
  </board-manifest-list>
  <app-manifest-list>
    <app-manifest>
      <uri>https://github.com/Infineon/partner-ce-manifest/raw/master/partner-ce-manifest-
    fv2.xml</uri>
    </app-manifest>
  </app-manifest-list>
  <middleware-manifest-list>
    <middleware-manifest dependency-url="https://github.com/Infineon/partner-mw-manifest/raw/
    master/partner-mw-dependencies-manifest.xml">
      <uri>https://github.com/Infineon/partner-mw-manifest/raw/master/partner-mw-manifest-
    fv2.xml</uri>
    </middleware-manifest>
  </middleware-manifest-list>
</super-manifest>
```

This field can be skipped if there are no dependencies attached to the manifests.

5.17.5.7 Validating your manifests

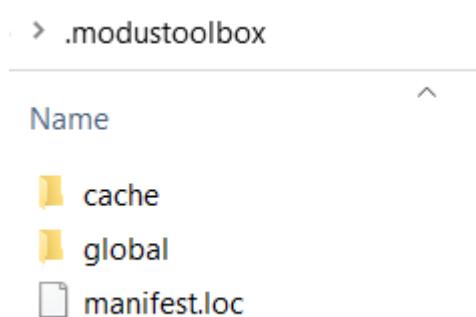
As a final check, all the manifests can be validated using any of the XML validators available online to ensure that you don't have any open XML sections, typos etc. If there are any errors in any of the manifest files, the content is not brought into the tools and will be ignored. If there are issues with values passed to the fields in the manifest, they can be noticed during the integration stage as explained in [Testing the manifest integration](#).

5 PSoC™ 6 application notes

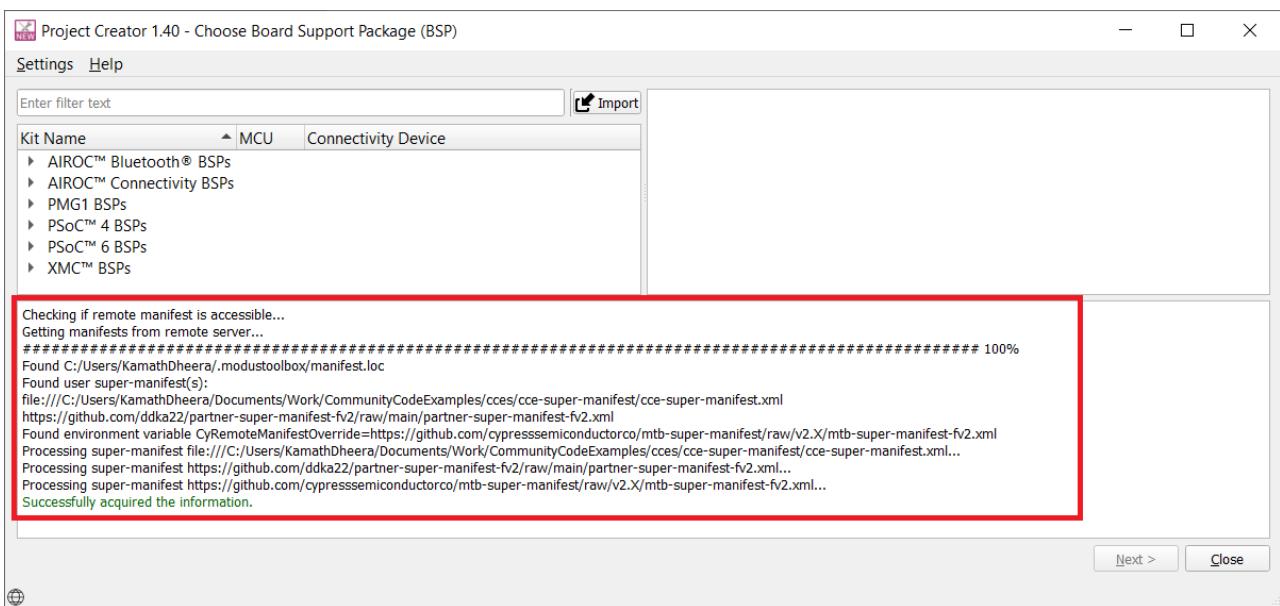
5.17.6 Testing the manifest integration

Once the content and manifests are created, the integration can be tested in ModusToolbox™ using the following steps:

- 1.** Navigate to the .modustoolbox folder (note the dot at the beginning of the folder name) which is located in your user profiles directory where ModusToolbox™ is installed by default. Sometimes this folder may be hidden, so unhide it in the file explorer view options to find it. For example, c:/Users/Username/.modustoolbox
- 2.** Add a file to this folder named manifest.loc.

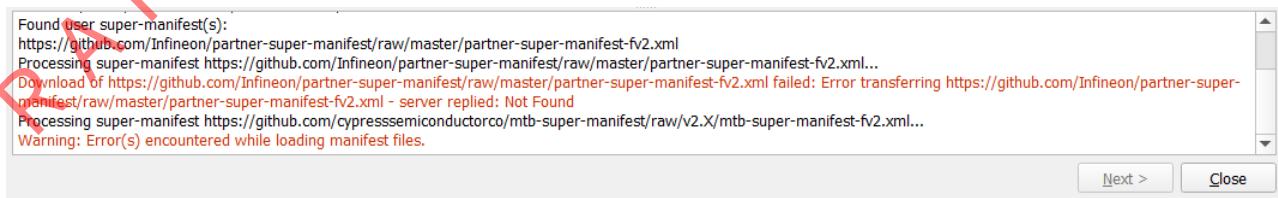


- Use an editor of your choice to edit the manifest.loc file to add the link to your super manifest such as <https://github.com/Infineon/partner-super-manifest/raw/master/partner-super-manifest-fv2.xml>.
- 3.** Run the Project Creator tool. It will list all the super manifests it is processing to bring the content in as shown below:

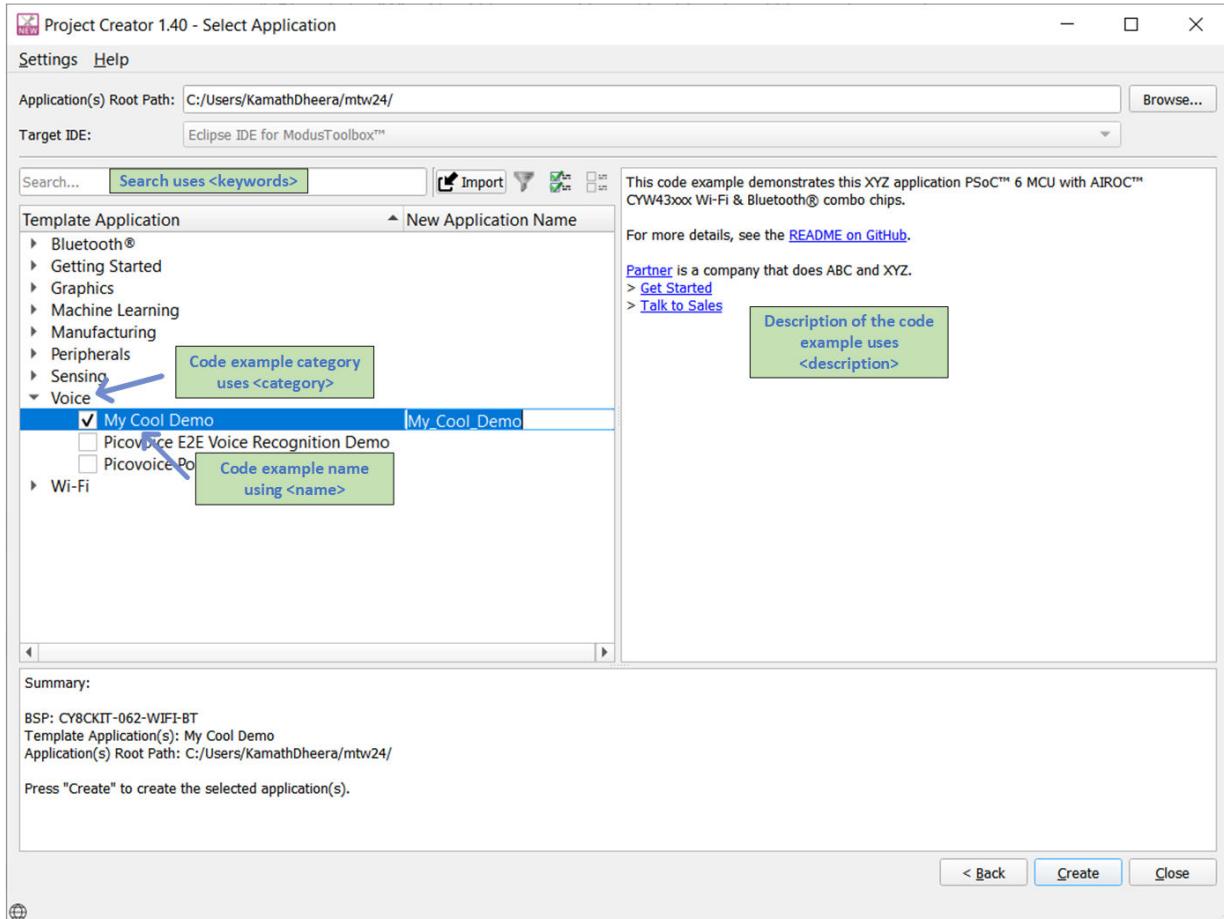


Note: If the tools were open previously, you will have to close and open them again for the manifest.loc changes to take effect. If any issues are encountered, Project Creator will display an error indicating the loading of the manifests failed as shown below:

5 PSoC™ 6 application notes



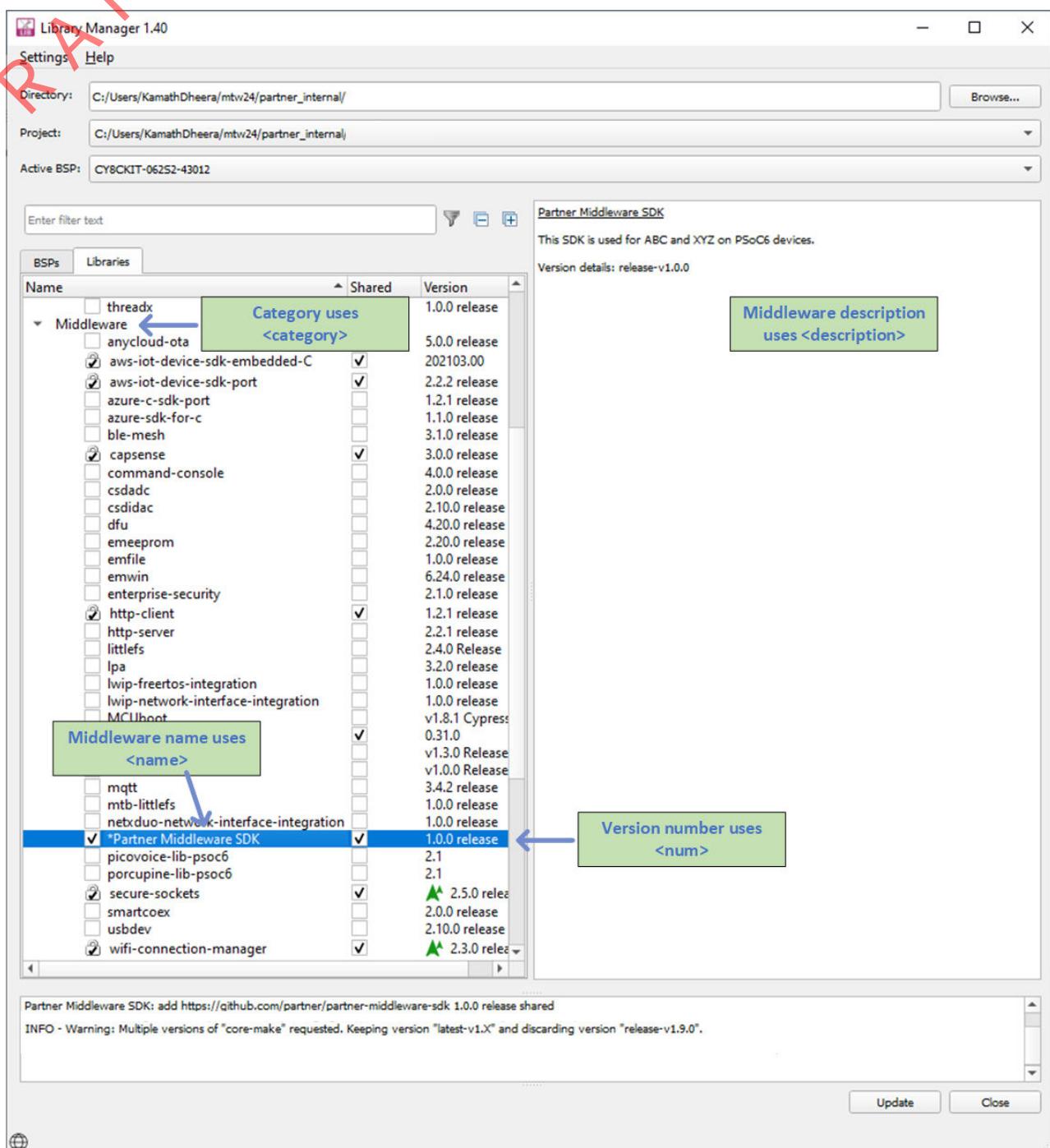
- You should be able to see the BSPs and code examples that you added being displayed in their respective tab and category.



- Create an application that uses your solution.
- Run the Library Manager tool.
- You should be able to see the middleware that you added being displayed in its respective tab and category.

5 PSoC™ 6 application notes

DRAFT



If you are able to see all the content displayed, you now have a successful setup in place to update your software content and make it available to ModusToolbox™ users seamlessly.

5.17.6.1 Testing the dependency manifest integration

As discussed previously, the middleware and BSPs can have dependencies on other libraries that are specified using the dependencies manifest. The expectation is that whenever any code example uses the .mtb file of the middleware or BSP, the corresponding dependent libraries are brought in automatically by ModusToolbox™ tools without the users needing to add them manually.

Follow these steps to check if the dependent libraries are being brought in:

- In your code example, add the .mtb file to the deps folder that points to your middleware or BSP that has dependent libraries.

5 PSoC™ 6 application notes

- ~~DRAFT~~
2. Push these changes to the upstream server and create a release tag. Update the code example manifest to add support for this new release.
 3. Open Project Creator and select the code example. Click **Create**.
 - In the Project Creator log, you should see the following sets of lines that indicate the resolution of the dependencies:

```
Resolving dependencies...
Checking if remote manifest is accessible...
Getting manifests from remote server...
Found C:/Users/User/.modustoolbox/manifest.loc
Processing super-manifest https://github.com/Infineon/partner-super-manifest/raw/main/mtb-
super-manifest-fv2.xml...
Successfully acquired the information.

INFO - Warning: Multiple versions of "core-make" requested. Keeping version "latest-v1.X"
and discarding version "release-v1.9.0".
C:/Users/User/mtw24/partner_internal/Partner_Demo/libs/abstraction-rtos.mtb was added
C:/Users/User/mtw24/partner_internal/Partner_Demo/libs/capsense.mtb was added
C:/Users/User/mtw24/partner_internal/Partner_Demo/libs/clip-support.mtb was added
C:/Users/User/mtw24/partner_internal/Partner_Demo/libs/connectivity-utilities.mtb was
added
C:/Users/User/mtw24/partner_internal/Partner_Demo/libs/core-lib.mtb was added
C:/Users/User/mtw24/partner_internal/Partner_Demo/libs/core-make.mtb was added
C:/Users/User/mtw24/partner_internal/Partner_Demo/libs/cy-mbedtls-acceleration.mtb was
added
C:/Users/User/mtw24/partner_internal/Partner_Demo/libs/freertos.mtb was added
C:/Users/User/mtw24/partner_internal/Partner_Demo/libs/lwip.mtb was added
C:/Users/User/mtw24/partner_internal/Partner_Demo/libs/mbedtls.mtb was added
C:/Users/User/mtw24/partner_internal/Partner_Demo/libs/mtb-hal-cat1.mtb was added
C:/Users/User/mtw24/partner_internal/Partner_Demo/libs/mtb-pdl-cat1.mtb was added
C:/Users/User/mtw24/partner_internal/Partner_Demo/libs/psoc6cm0p.mtb was added
C:/Users/User/mtw24/partner_internal/Partner_Demo/libs/recipe-make-cat1a.mtb was added
C:/Users/User/mtw24/partner_internal/Partner_Demo/libs/secure-sockets.mtb was added
C:/Users/User/mtw24/partner_internal/Partner_Demo/libs/whd-bsp-integration.mtb was added
C:/Users/User/mtw24/partner_internal/Partner_Demo/libs/wifi-connection-manager.mtb was
added
C:/Users/User/mtw24/partner_internal/Partner_Demo/libs/wifi-host-driver.mtb was added
C:/Users/User/mtw24/partner_internal/Partner_Demo/libs/wifi-mw-core.mtb was added
C:/Users/User/mtw24/partner_internal/Partner_Demo/libs/wpa3-external-supplicant.mtb was
added
Dependencies resolved.
```

4. Once the project is created, check if all the dependent libraries are available in the `mtb_shared` directory. If all the libraries are available, the dependencies integration is a success.

5 PSoC™ 6 application notes~~DRAFT~~
5.17.6.2 Out-of-the-box testing

Now that the content is visible in ModusToolbox™ tools, test if the content is being downloaded correctly and works as expected. During the creation of the code example, Project Creator should import all the necessary libraries and the middleware automatically. The code example should build out-of-the-box. To test this, do the following (it is assumed you have added your super manifest file to `manifest.loc` file):

1. Run Project Creator.
2. Select the code example you added in your code example manifest.
3. Click **Create**. The code example will be cloned from your Git repository and all the dependencies will be brought in.
4. Now, build your code example and make sure it works as expected. If there are modifications to be made for the build to be successful, document the steps in the `ReadMe.md` file.
5. Program the code on the hardware to verify the functionality.

5 PSoC™ 6 application notes**5.17.7 Integrating into ModusToolbox™**

Once the integration of the content and manifest has been tested, do the following to get the content integrated formally with ModusToolbox™:

- 1.** Reach out to Infineon by creating a ticket in the case system as documented in [Technical support](#). Provide a brief description indicating that the ticket is a request for your content to be integrated along with links to your manifests.
- 2.** Infineon technical support will get in touch with you and support you with the integration. The manifests will be reviewed and content will be tested to ensure that everything works as expected in ModusToolbox™.
- 3.** Any concerns with the manifest or the content will be raised in the ticket. Partners must address all the concerns and provide an update of the resolution in the ticket.
- 4.** Once the technical support team finds the content and manifest ready to be integrated, the ModusToolbox™ super manifest will be updated to point to the partner manifests.
- 5.** The update pertaining to the integration will be provided via the ticket. Partners can validate the integration and close the ticket if everything looks good.

Partners can continue to manage their content via their own manifests. The partners are free to manage, update, and create new content using their manifests without any need to contact Infineon. However, care should be taken while updating manifests to prevent errors. Any update to the content or manifests should follow the guidelines described in section [Updating your content](#).

Steps 1 through 5 should be repeated only when new manifest files need to be integrated. For further queries, contact Technical Support by creating a ticket (see [Technical support](#)).

5 PSoC™ 6 application notes**5.17.8 Updating your content**

~~DRAFT~~ Now that the initial version of the content and manifest is already integrated into ModusToolbox™, any update should be handled with care. **Infineon reserves the right to temporarily pull down any content or manifest that does not work as intended or affects the user experience in ModusToolbox™ until it is rectified.**

When updating your content, the following steps should be followed.

1. [Clone repositories](#)
2. [Create a topic branch](#)
3. [Update and test your content](#)
4. [Merge into mainline](#)
5. [Create release package](#)
6. [Update corresponding manifest](#)
7. [Update dependency manifest](#)
8. [Testing the integration](#)

Note: *Partners must strictly follow the guidelines to update the content and manifests to ensure it doesn't break the user experience in ModusToolbox™. Infineon will provide an indication via proper communication channels to partners to rectify their content when such cases arise. If the guidelines are not followed and these problems keep recurring, the content and manifests will be permanently taken down after three such occurrences.*

5.17.8.1 Clone repositories

Clone the repository of the content you want to update. Use `git clone` command to clone your repository.

5.17.8.2 Create a topic branch

Create a new topic branch to work on the updates. This prevents any accidental code being pushed to the mainline that can affect your users. Use `git checkout -b <branch_name>` command to create a new topic branch.

5.17.8.3 Update and test your content

Update the source code and documentation. Make sure that the changes are tested thoroughly. If the source code is updated, the functional tests should be conducted to verify the working on the hardware.

For content that has dependencies on others, the asset that has the dependencies should also be tested. If the asset breaks, the asset should be updated to work with the updated dependencies. For example, a code example has dependencies on middleware. If the middleware is updated, the code example should also be tested to verify if it works with the updated middleware. If not, the code example should be updated to work with the updated middleware.

5.17.8.4 Merge into mainline

Merge the fully tested content into the main branch. Any merge conflicts that may arise as a result of this action should be resolved. If this is a dual-stage setup, deploy the contents into production.

5.17.8.5 Create release package

The release tags should be created for any new update to the content. See the table in [Creating the release package](#) to understand how to update the version number.

5 PSoC™ 6 application notes

5.17.8.6 Update corresponding manifest

Once the release tag is created, the corresponding manifest should be updated based on the content being updated. For example, if the code example is updated and a new release tag “release-v2.0.0” is created, the CE manifest should be updated to reflect this change as highlighted below:

Code Listing 24

```
<apps version="2.0">
  <app keywords="psoc6,partner,my,cool,demo">
    <name>My Cool Demo</name>
    <category>Voice</category>
    <id>partner-mtb-example-my-cool-demo</id>
    <uri>https://github.com/partner/partner-mtb-example-my-cool-demo</uri>
    <description><![CDATA[This code example demonstrates this XYZ application PSoC™ 6 MCU with AIROC™ CYW43xxx Wi-Fi & Bluetooth® combo chips.
      <br><br>For more details, see the <a href="https://github.com/partner/partner-mtb-example-my-cool-demo/blob/master/README.md">README on GitHub</a>. <br><br><a href="https://partner-webpage.com/">Partner</a> is a company that does ABC and XYZ.<br> > <a href="https://docs.partner-doc-page.com/">Get Started</a><br> > <a href="https://partner-webpage.com/contact/">Talk to Sales</a>]]></description>
    <req_capabilities>psoc6</req_capabilities>
    <versions>
      <version flow_version="2.0" tools_min_version="2.4.0" req_capabilities_per_version="bsp_gen3">
        <num>2.0.0 release</num>
        <commit>release-v2.0.0</commit>
      </version>
      <version flow_version="2.0" tools_min_version="2.4.0" req_capabilities_per_version="bsp_gen3">
        <num>1.0.0 release</num>
        <commit>release-v1.0.0</commit>
      </version>
    </versions>
  </app>
</apps>
```

Note: Only the CE, BSP, MW, and corresponding dependency manifests will be updated whenever there is an update. The super manifest will never be modified unless there is a new manifest file that should be pointed to. In that case, you must follow the steps in [Integrating into ModusToolbox™](#) to get your super manifest changes integrated into the ModusToolbox™ super manifest.

5 PSoC™ 6 application notes

5.17.8.7 ~~DRAFT~~ Update dependency manifest

If the update to the content requires any newer versions of the dependent libraries, the dependency manifest should be updated to reflect this change. For example, if the middleware is updated to release-v2.0.0 and it uses newer version latest-v3.x of the wifi-mw-manager library, the middleware dependency manifest should reflect that change as highlighted below:

Code Listing 25

```
<dependencies>
  <depender>
    <id>partner-middleware-sdk</id>
    <versions>
      <version>
        <commit>release-v2.0.0</commit>
        <dependees>
          <dependee>
            <id>wifi-connection-manager</id>
            <commit>latest-v3.X</commit>
          </dependee>
          <dependee>
            <id>http-client</id>
            <commit>latest-v1.X</commit>
          </dependee>
        </dependees>
      </version>
      <version>
        <commit>release-v1.0.0</commit>
        <dependees>
          <dependee>
            <id>wifi-connection-manager</id>
            <commit>latest-v2.X</commit>
          </dependee>
          <dependee>
            <id>http-client</id>
            <commit>latest-v1.X</commit>
          </dependee>
        </dependees>
      </version>
    </versions>
  </depender>
</dependencies>
```

5.17.8.8 Testing the integration

Follow the steps described in [Testing the manifest integration](#) to validate your manifests and check if the content is working as expected post integration.

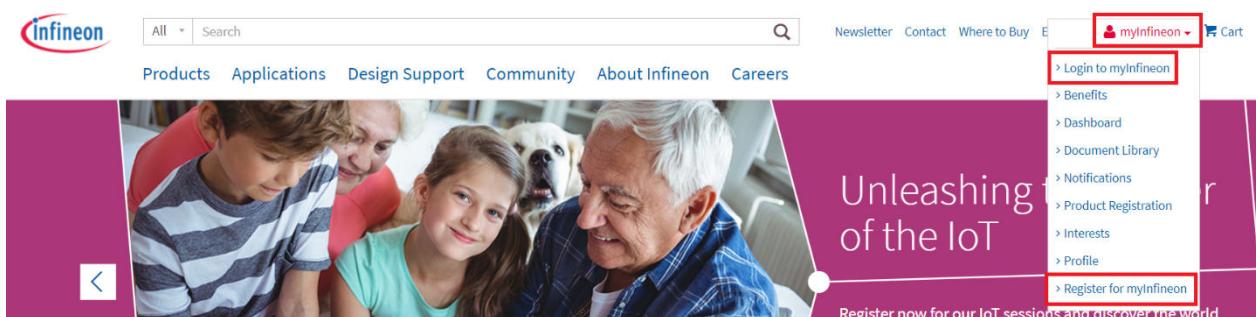
5 PSoC™ 6 application notes

5.17.9 Technical support

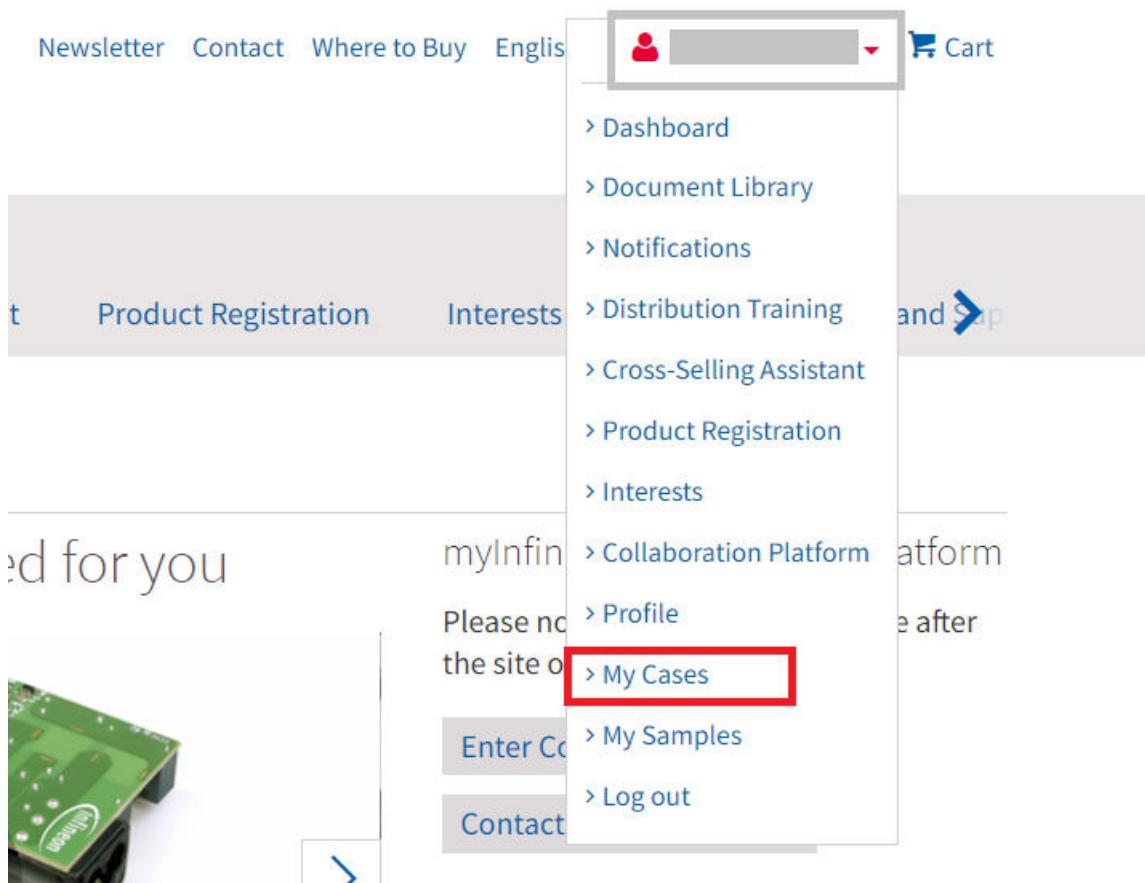
This section documents how you can contact Technical Support when you face any issues or have questions about your implementation. Infineon offers a case system to create tickets to seek technical support. Additionally, once you have successfully tested your manifest implementation, reach out to Infineon via the technical support channel to request the manifests to be integrated into ModusToolbox™.

Here are the steps to contact technical support:

1. Navigate to <https://www.infineon.com/>.
2. In the right-top corner, select the drop-down that reads “myInfineon” and click Login to myInfineon. For new users, click Register for myInfineon to create your account and then login. The steps are very straightforward.



3. Once logged in, click **MyCases** in the drop-down under your profile.



4. On the MyCases webpage, choose **Technical Support** option available inside the “Create a MyCase” box as highlighted below:

5 PSoC™ 6 application notes

DRAFT

Create a MyCase

- | | |
|--|--|
| Customer Commitment
Customer Service
Cyber\Product Security
Distributor Operations
Expedite/Change Order
Failure Analysis | Product Quality
Quality Documentation
QTP/RMP Analysis Request
RMA Request
Tech Support Forum
Technical Support |
|--|--|

Submit MyCase for Questions/Problems related to new designs, datasheets, timing, functionality, device compatibility, and general applications support.

- You will be directed to a new webpage with a form to fill out the details of your query. The table below lists the various fields and values that need to be entered.

Field	Description	Example
Product	Select the product you were working on which led to the query. Use the search icon next to this field to search for part numbers or product family that this query is closest to. If your query does not relate to any product, click “I do not know the product”.	CY8C6247BZI9-D54
Inquiry type	Select the type of query. Choose between different options available in the drop-down. Use “CY Software Tools” for queries related to this document.	CY Software Tools
Link Opportunity	Specify the opportunity type. This is optional and need not be specified. Leave it blank.	
Priority	Specify the priority of the request. Choose between low, medium, and critical based on how much the request affects your work and how soon you want the resolution.	Low
End customer	Specify the account associated with the customer creating the ticket. Use the partner account issued to you as part of the partner program.	CompanyName
On behalf of email	Provide your email address.	John.Doe@partner.com

5 PSoC™ 6 application notes

Field	Description	Example
Project	Specify the project related to the issue. Use “MODUSTOOLBOX & FRIENDS” category for all issues related to this document.	MODUSTOOLBOX & FRIENDS
Subject	Specify a summary of the issue or query.	<i>Issue with super manifest</i>
Description	Describe the issue in detail. Make sure that you provide relevant screenshots under “Attachment” to illustrate your issue better.	<i>Hi, I have problems implementing super manifest. Here is what I implemented. Please see attached file. I see the following issues.</i>
Attachment	Choose files or images to illustrate the problem better.	

Note: If you have successfully implemented the manifests, mention the same in the description and request the Technical Support team to have them integrated into ModusToolbox™. The Technical Support team will review the manifests and have them integrated. Post integration, updates will be provided on the ticket.

5. Click **Create MyCase** to submit your request. A ticket will be generated and the status can be viewed under “My Created Cases” section.

My Created Cases						
MyCase No	Type	Inquiry Type	Subject	Status	Open	Last Response
00655509	Tech Support	User Application Specific	MSN Query	Waiting on Customer	9/8/2021 12:09 pm	2/9/2021 6:48 pm
00716953	Tech Support	CY Software Tools	Test	Open	22/7/2022 4:29 am	

- A technical support engineer will be assigned to the thread within the next 24 hours. They will reach out to you via the ticket by adding their comments. Email notifications will be received whenever there is any activity on the ticket.

6. Use the **Interactions** box to interact with the technical support engineers. If your issue is resolved, use the **Case Resolved – Close Case** button to close the case.

The screenshot shows the "Interactions" interface for a specific case. It includes fields for "Subject" (containing "[MyCase#: 00716953] - Test"), "Interaction Type" (a dropdown menu), "Message" (a large text area), "Additional Recipient(s)" (an input field), and "Attach a File" (a file upload section). At the bottom are buttons for "Add Comment", "Escalate", and "Case Resolved - Close Case".

7. Use steps 1 through 8 to create a new case whenever you face any issues in the future.

5 PSoC™ 6 application notes~~DRAFT~~
5.17.10 Summary

This application note has shown how partners can get onboarded into the “ModusToolbox™ & Friends” program and contribute content to the ModusToolbox™ ecosystem. As mentioned earlier, this provides partners the ability to showcase content directly to developers using ModusToolbox™ to create interesting applications.

[Appendix A – Partners integrated into ModusToolbox™](#) in this document features a list of partners who have been successfully integrated as part of this program and reference links to their content.

5 PSoC™ 6 application notes**5.17.11 Appendix A – Partners integrated into ModusToolbox™**

This appendix features a representative partner successfully integrated into ModusToolbox™ as part of the “ModusToolbox™ & Friends” program.

5.17.11.1 Memfault

[Memfault](#) is the first cloud-based observability platform for connected device debugging, monitoring, and updating, which brings the efficiencies and innovation of software development to hardware processes.

Memfault, as part of “ModusToolbox™ & Friends”, leveraged the benefits of the program and developed content to make remote debugging and monitoring accessible to developers using ModusToolbox™. With the combination of Infineon’s powerful PSoC™ 6 MCUs and their proprietary firmware SDK, they demonstrated an innovative solution on how to approach debugging.

Here are some quick links to the content and manifests developed by Memfault (for reference only):

- [Memfault Super Manifest](#)
- [Memfault Middleware Manifest](#)
- [Memfault Code Example Manifest](#)
- [Memfault Middleware](#)
- [Memfault Code Example](#)

5 PSoC™ 6 application notes**5.17.12 Revision history**

Document version	Date of release	Description of changes
**	2022-07-29	Initial release

5.18 AN235279 Performing ETM and ITM trace on PSoC™ 6 MCU**About this document**

-
- 1
- 8

Scope and purpose

The application note introduces the trace features of the PSoC™ 6 MCU and the supporting software tools. This application note helps you get started with performing instruction (ETM) and instrumentation (ITM) tracing on PSoC™ 6 MCUs. A sample project is configured using ModusToolbox™ and exported to third party tools like IAR Embedded Workbench and Keil µVision to perform trace. The application note also guides you to more features of trace and other resources available online to accelerate your learning.

If you are new to PSoC™ 6 MCU and ModusToolbox™ software environment, see [AN228571 - Getting started with PSoC™ 6 MCU on ModusToolbox™ software](#).

Intended audience

The application note is intended for advanced engineers who want to use the trace capabilities of the PSoC™ 6 MCU.

5 PSoC™ 6 application notes**5.18.1 Introduction****5.18.1.1 What is trace?**

As per Arm®'s definition, 'trace' refers to the process of capturing data that illustrates how components in a design are operating, executing, and performing. In simple words, trace helps you capture and visualize the operations that are occurring inside the MCU, generally in a non-intrusive way.

There are several types of trace; for each type, usually a separate trace generation component is implemented in the MCU. Trace can be broadly classified as follows:

- Instruction trace
- Data trace
- Instrumentation trace
- System trace

Although the scope of this application note is limited to instruction and instrumentation trace, the approach can be extended to perform data and system trace as well.

Instruction trace generates information about the instruction execution of a core or processor. **Embedded Trace Macrocell (ETM)** is one such example for instruction trace source.

Instrumentation trace outputs data collected from several hardware (for example, Data Watchpoint and Trace unit - DWT) and software sources (for example, 32-bit stimulus registers). Data from the instrumentation trace can be output using printf-style debugging. **Instrumentation Trace Macrocell (ITM)** is used to capture the instrumentation trace data.

5.18.1.2 Why is trace important?

Designing a complex embedded system efficiently depends directly on the ability to precisely debug the issues that appear during development. The trace technique has the major benefit of its being non-intrusive compared to other debugging techniques. It means that trace can be used to fetch all the instructions running in the core without affecting the actual application flow. Not only instructions, other system-level information and data can be analyzed without halting the processor or adding extra lines of code.

Along with the cycle count information and timestamps, the trace data can also be used to profile the code and measure performance at function level.

See the [Non-intrusive debugging with ETM trace](#) article from IAR Systems for a good illustration of the importance of trace.

5.18.1.3 Overview of subsequent chapters

The subsequent chapters in this application note will speak about the overview of Arm® trace architecture and its implementation in PSoC™ 6 MCUs. The hardware and software requirements for performing trace on PSoC™ 6 MCUs are listed. Later sections describe how to import an existing PSoC™ 6 MCU project in ModusToolbox™ and enable trace from ModusToolbox™. Finally, steps on exporting a ModusToolbox™ project, editing the debugger scripts, and performing trace on third party tools like IAR Embedded Workbench and Keil µVision are shown.

~~DRAFT~~ 5 PSoC™ 6 application notes

5.18.2 General and PSoC™ 6 MCU Arm® trace architecture

5.18.2.1 General trace architecture

Trace components can be mainly classified into three types:

- **Trace Source**: A component which generates trace data; for example: ETM, ITM
- **Trace Sink**: A component which stores or outputs the trace data; for example: Embedded Trace Buffer (ETB), Embedded Trace FIFO (ETF), Trace Port Interface Unit (TPIU), Serial Wire Output (SWO)
- **Trace Link**: A component which links trace or non-trace components together; for example: Funnel, Replicator, Cross Trigger Interface (CTI).

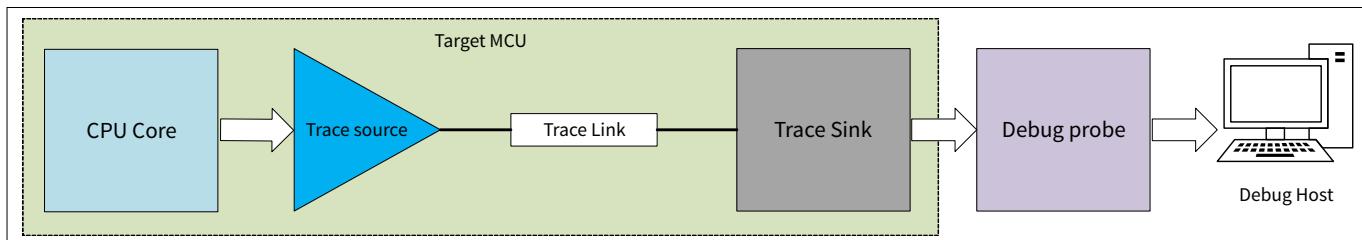


Figure 597 Simplified general trace architecture

5.18.2.1.1 Trace Source

Although there are many trace source components, the scope of this application note is limited to ETM and ITM components.

Embedded Trace Macrocell (ETM)

The ETM component allows instruction and data trace. The ETM block is configurable during chip design; you can omit data trace if not required.

At a high level, ETM does not generate a trace packet for every instruction the CPU executes. It only outputs information about the instruction flow (jump or no jump) and sometimes outputs the full destination address (when there is a branching instruction). The debug host will generally have a copy of the application image; therefore, using the data from the ETM component, the complete program execution can be reconstructed. As CPU speeds are usually higher, the ETM block needs to compress the instruction execution data and packetize it before sending it to a trace sink.

Between the ETM block and a trace sink, a FIFO buffer is usually provided to allow sufficient time for the trace sink to process and route the trace data.

ETM data also includes time-stamps; this can be used to determine the time consumed by code or functions. Function profiling is very useful in optimizing the regions of your code which is consuming a lot of CPU bandwidth.

Instrumentation Trace Macrocell (ITM)

The ITM component is an application-driven trace source. The data for ITM can come from several sources.

- Software source – The application can write directly to the ITM stimulus registers to generate packets. An example use case is sending out raw ADC data, which can be plotted and visualized at the debug host.
- Hardware source – The DWT block has data watchpoints, data trace, debug event, and profiling counters. The data from these debug systems can be routed through ITM.
- Timestamps – A counter in the ITM provides a timestamp for each trace packet.

5 PSoC™ 6 application notes

~~DETAILED~~ Micro Trace Buffer (MTB)

The MTB component allows basic instruction trace. During a trace operation, the debugger can configure the MTB to allocate a small portion of the SRAM as a trace buffer for storing the trace information. The SRAM can be a dedicated or a shared one (system SRAM used by the CPU). When a branching instruction occurs or if the program flow changes due to interrupts, the MTB stores the source and destination PC (program counter) information. The MTB is mostly used in circular buffer mode: when the allocated memory is full, the oldest branch information is overwritten by a new branch information.

5.18.2.1.2 Trace Sink

Similar to the trace source, there are many trace sink components. The scope of this document is limited to ETB, ETF, TPIU, and SWO.

Embedded Trace Buffer (ETB)

The ETB is a dedicated SRAM that stores generated trace data on-chip for later retrieval and analysis. The SRAM acts like a circular buffer that wraps when the buffer size limit is reached. Buffer wrapping works by replacing the oldest trace data with the newest data.

Embedded Trace FIFO (ETF)

The Embedded Trace FIFO (ETF) contains a dedicated SRAM that can be used as either a circular buffer, a hardware FIFO, or a software FIFO. In circular buffer mode, the ETF has the same functionality as the ETB. In hardware FIFO mode, the ETF is typically used to smooth out fluctuations in the trace data. In software FIFO mode, on-chip software uses the ETF to read out the data over the debug AMBA Peripheral Bus (APB) interface.

Trace Port Interface Unit (TPIU)

The TPIU routes the trace data to external pins. A debugger is connected to these external pins to capture the trace data. The TPIU also adds source identification information into the trace stream so that trace can be re-associated with its trace source.

Usually, four data pins and one clock pin are associated with the TPIU component. This is a design-time configuration and can be changed per need. If the TPIU block has connection to four data pins, it is not necessary to use all the four pins to output the trace data. In this case, the TPIU can be configured to be used in 1-bit, 2-bit, or 4-bit mode.

Serial Wire Output (SWO)

The trace data from the source is directly passed to an external debugger using a single-wire output called SWO. Owing to the trace bandwidth required, the single-pin SWO is not suitable for outputting the ETM trace data; it is mainly used to pass the ITM data.

5.18.2.1.3 Trace Link

Funnel

The funnel merges multiple AMBA Trace Bus (ATB – bus that carries data from trace sources) into a single ATB. Then the single ATB can be routed to a trace sink, replicator, or another trace funnel.

~~5 PSoC™ 6 application notes~~

~~Replicator~~

The ATB that comes out of the funnel can be split into multiple ATBs using a replicator. This allows the trace data to be routed to multiple sinks.

~~Cross Trigger Network (CTI)~~

The CTI is used to generate and route triggers between different trace components. For example, the ETB can send a trigger to the ETM block to halt the CPU when the buffer is full.

5.18.2.2 Trace output

As discussed earlier, trace data is bandwidth-intensive, and therefore needs compression/encoding before converting to packets. Therefore, the trace data is not directly in a human-readable format. The trace data when captured by a debugger is decompressed/decoded and processed to convert it to a human-readable format. Sometimes, when transmitting raw data through the ITM, the trace packets can directly have the raw data without encoding.

The trace data can be captured in two ways by a debugger:

- On-chip capture
- Off-chip capture

5.18.2.2.1 On-chip capture

In this scenario, the trace data is usually stored in the ETB. At particular points during debugging, the external debugger performs operations to extract the on-chip trace data using a 2-pin serial wire debug (SWD).

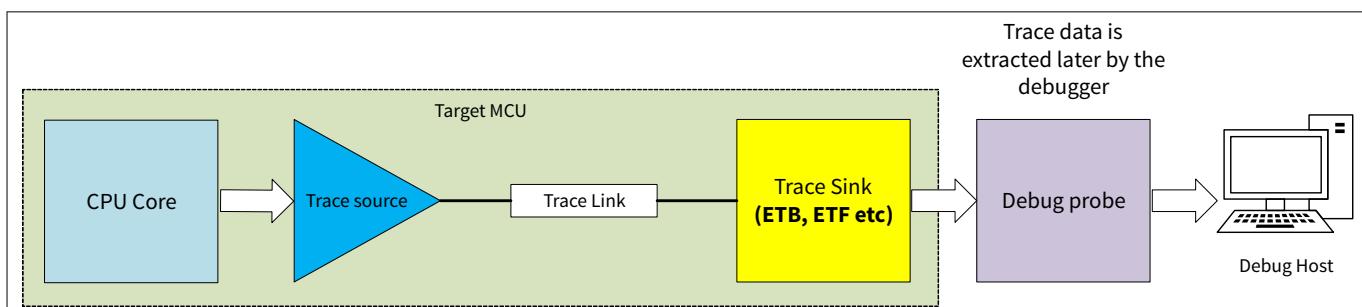


Figure 598 Trace architecture for on-chip capture

5.18.2.2.2 Off-chip capture

In this scenario, the trace data is routed to an external debugger in real time using the TPIU and SWO pins. The debugger then processes the data and displays it in a human-readable format. The scope of this application note is mainly to learn this mode of trace capturing.

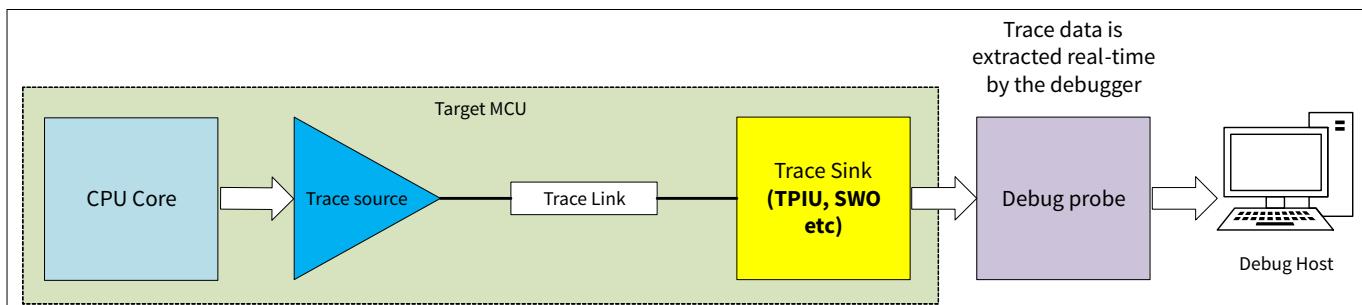


Figure 599 Trace architecture for off-chip capture

~~5 PSoC™ 6 application notes~~

~~5.18.2.3 Trace infrastructure examples~~

Trace components are highly design-configurable; the choice of components to use is also left to the MCU vendors. This section shows a few trace infrastructure examples that can be implemented during design.

~~5.18.2.3.1 Single-core, off-chip ETM trace example~~

In this example, the ETM is used to generate both instruction and data traces for the core. The ETF is used to buffer the data before sending the data to the TPIU component. The TPIU component then forwards the data to the external debugger in real time using the data and clock TPIU pins.

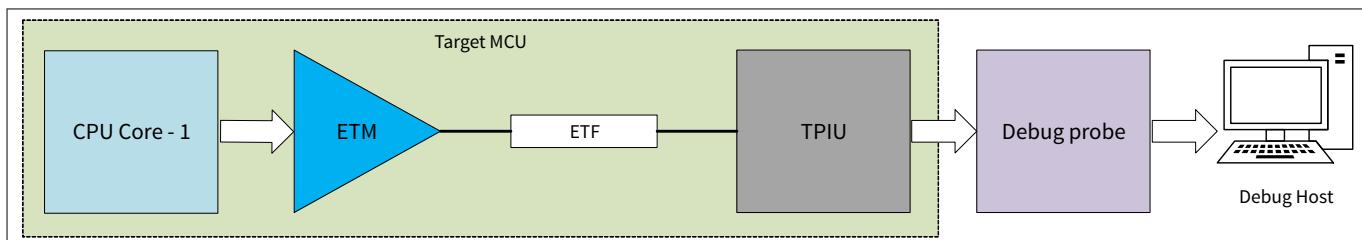


Figure 600 Trace architecture for single-core, off-chip ETM trace

~~5.18.2.3.2 Multi-core, off-chip ETM trace example~~

In this example ETM data from multiple cores is merged to one ATB using the funnel. The data from the funnel is then routed through the ETF, TPIU, and finally to the debugger.

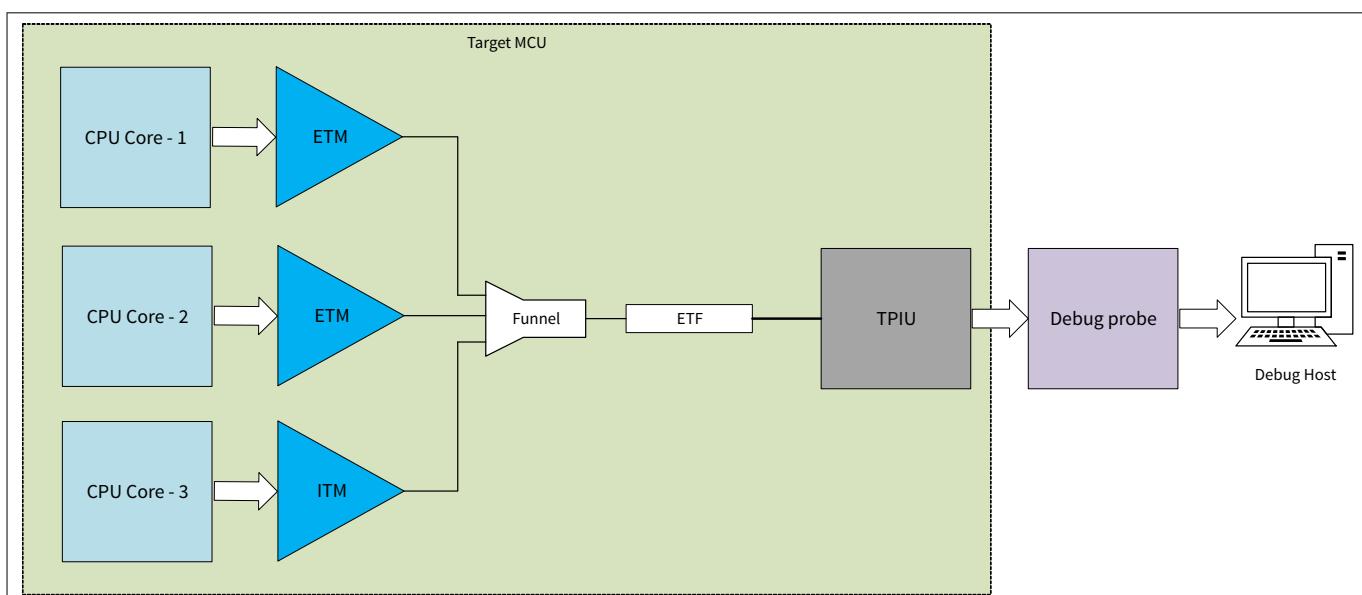


Figure 601 Trace architecture for multi-core, off-chip ETM trace

~~5.18.2.3.3 Multi-core, off-chip ETM CTI trace example~~

This example is similar to the previous multiple-core example, except that now triggers are routed between different trace components. For example, the ETF can send a halt request to the cores if the FIFO memory starts to overflow. The halt request halts the cores, which allows the debugger to collect the trace data; then the ETF sends a resume request to start the cores back once the FIFO has free space.

5 PSoC™ 6 application notes

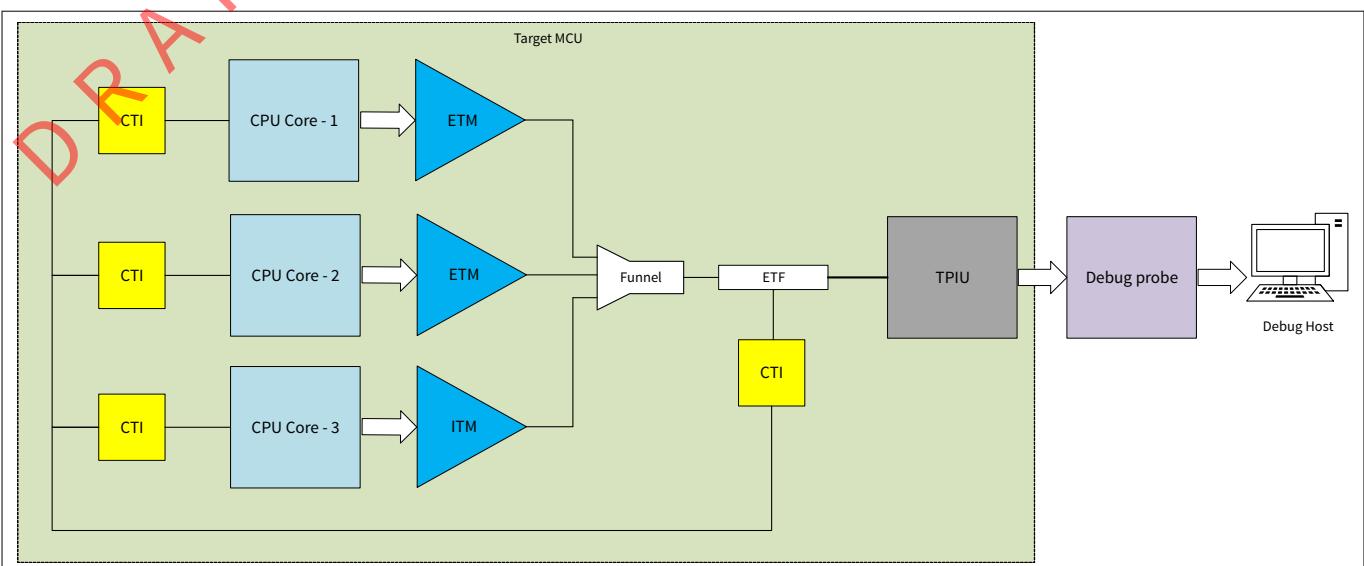


Figure 602 Trace architecture for multi-core, off-chip ETM trace with CTI

5.18.2.4 PSoC™ 6 MCU trace infrastructure

PSoC™ 6 MCU is a dual-CPU microcontroller with Arm® CM4 and CM0+ cores.

The CM4 core supports a 4-bit ETM and an ITM as trace sources. It supports a TPIU port (4 data pins, 1 clock pin) and a SWO pin for outputting the trace data to the external debugger. The SWO pin is multiplexed with the JTAG interface; both cannot be used simultaneously. The TPIU pins can be routed to multiple I/O ports on PSoC™ 6 MCU; see the [device datasheet](#) for details.

The CM0+ CPU supports the MTB with 4-KB dedicated SRAM. This application note does not cover MTB tracing.

PSoC™ 6 MCU also has an Embedded Cross Trigger (also called CTI) for synchronized debugging and tracing of both CPUs.

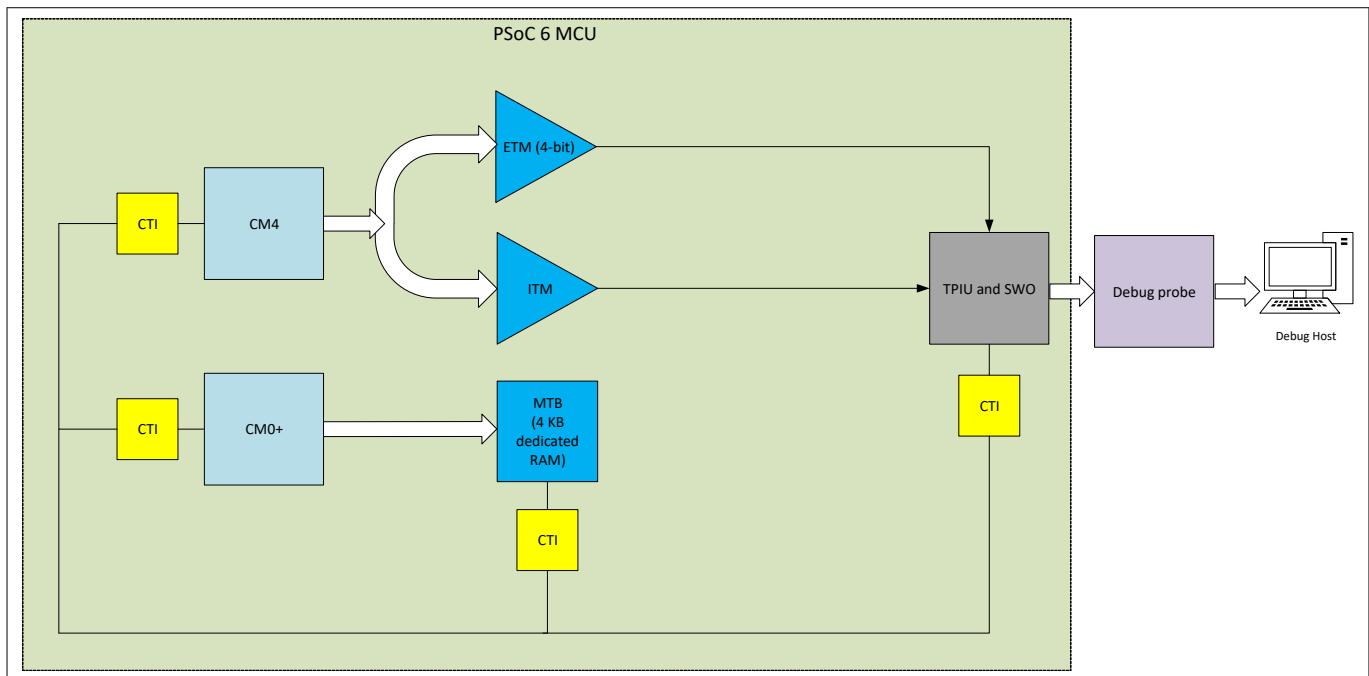


Figure 603 PSoC™ 6 MCU trace architecture

~~DRAFT~~ 5 PSoC™ 6 application notes

5.18.3 Hardware and software requirements

This section will list the hardware and software requirements to perform hands-on trace on PSoC™ 6 MCUs.

5.18.3.1 Hardware requirements

5.18.3.1.1 Trace probes (external debugger)

The flow in this application note uses two trace probes:

- I-jet Trace for Arm® Cortex® -M – The I-jet trace probe is used with IAR Embedded Workbench software tools.
- ULINKpro Debug and Trace Unit – The ULINKpro trace probe is used with Keil µVision software tools.

5.18.3.1.2 Development board

The flow in this application note uses the [CY8CEVAL-062S2 evaluation kit](#). Usually on the evaluation kits, due to limited I/Os, the trace pins are multiplexed with other peripherals. Open the [CY8CEVAL-062S2 schematics](#) file, search for the term “trace”. You will find a box named “Trace multiplexed pins” as shown below:

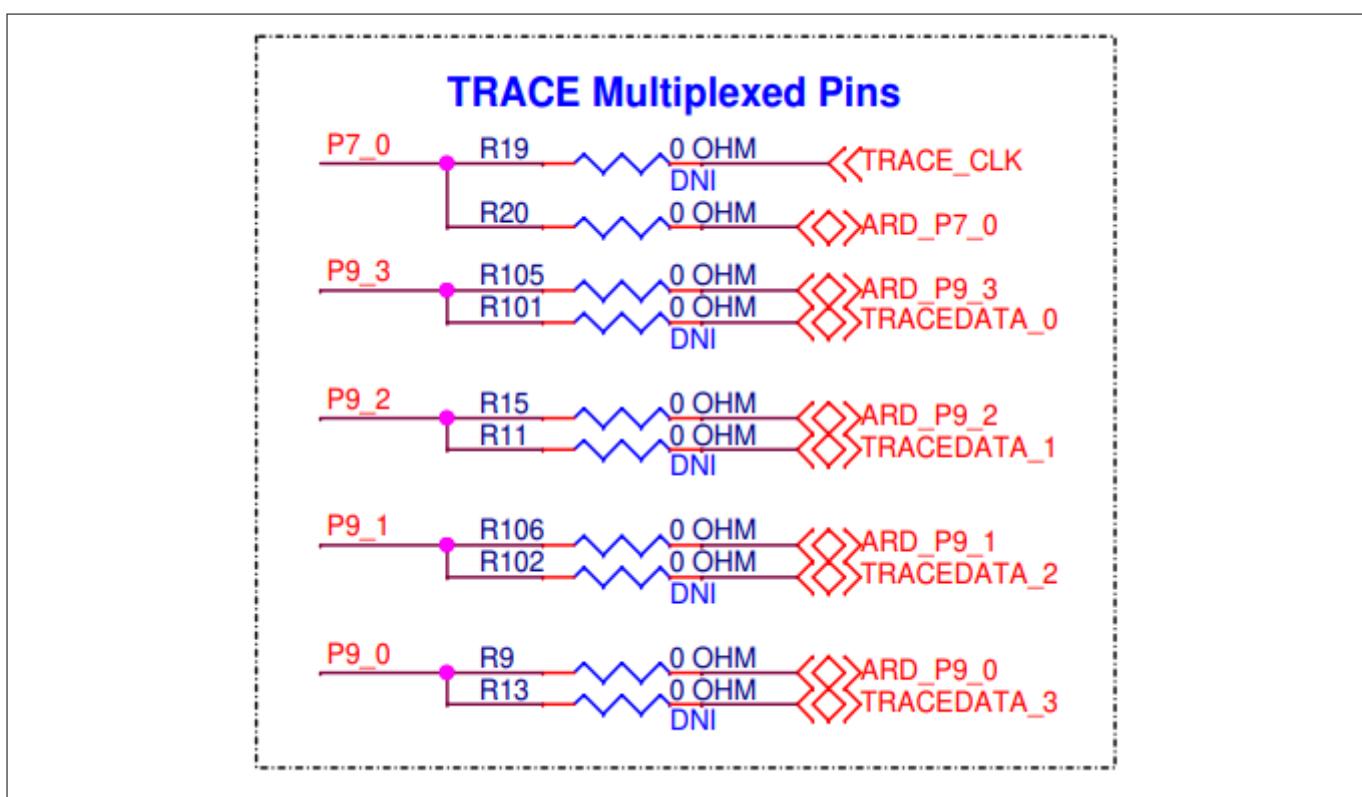


Figure 604 CY8CEVAL-062S2 kit trace pin schematics

In this kit, the trace pins are multiplexed to the Arduino port pins. The trace pins must be exposed to the J22 ETM header by doing the following:

1. Unload the resistances R20, R105, R15, R106, and R9.
2. Load zero-ohm resistances to R19, R101, R11, R102, and R13.
3. Solder on the ETM header (20-pin FRC male connector).

Note: The [CY8CKIT-062-BLE](#) and [CY8CKIT-062-WIFI-BT](#) evaluation kits do not need any hardware modification, the trace pins are brought out on to the 20-pin debug header out-of-the box. You must

5 PSoC™ 6 application notes

~~DRAFT~~
perform the hardware modification steps for any other evaluation kit/custom board to make sure that the trace pins are properly brought out.

Note: The PSoC™ 63 Bluetooth® LE devices are not supported to perform ETM trace using ULINKpro. However, you can use other supported trace units (for example, I-jet Trace and J-Trace PRO) to perform ETM trace on the PSoC™ 63 Bluetooth® LE devices.

5.18.3.2 Software requirements

The flow in this application note uses the following software tools:

- [ModusToolbox™ software](#) – ModusToolbox™ 3.0 is required to create/import a project for PSoC™ 6 MCUs. The tool is also used to export the project to third party tools.
- [IAR Embedded Workbench for Arm®](#) – IAR EW (tested with v8.42.2) and I-jet trace can be used to perform trace on PSoC™ 6 MCUs.
- [Keil µ Vision](#) – µVision (tested with v5.36) and ULINKpro can be used to perform trace on PSoC™ 6 MCUs.

DRAFT
5 PSoC™ 6 application notes**5.18.4 Performing trace on PSoC™ 6 MCU****5.18.4.1 Creating/importing project using ModusToolbox™**

As ETM and ITM are supported only on the CM4 core of the PSoC™ 6 MCU, the application note will use a single-core project for trace demonstration. The following steps explain how to import the [PSoC™ 6 MCU Empty App](#) code example to Eclipse IDE for ModusToolbox™ and configure the project to reserve resources for trace:

1. Launch Eclipse IDE for ModusToolbox™ 3.0 or later and select a workspace.
2. From the **Quick Panel**, click **New Application**.
3. Once the project creator wizard opens, expand **PSoC™ 6 BSPs** and select **CY8CEVAL-062S2**. Click **Next**.
4. In the new window, expand **Getting Started** and select **Empty App**. Click **Create**.
5. Once the application is loaded in the Project Explorer, select the application folder. From the Quick Panel, launch **Device Configurator**.
6. In the Device Configurator, select the **System** tab. Select **Debug** under **Resources**.
7. Edit the debug parameters as shown in the following figure to reserve resources for performing trace. This step is required **only to reserve** the resources, so that they are not accidentally assigned to some other peripheral by the HAL hardware manager. The configuration of the same resources is actually done by the third party debugger scripts.

When selecting the trace pins and clock divider, make sure to pick the resources that are defined in the BSP and have aliases starting with `cyBSP_`. The pins defined in the BSP will match the pins as seen in the [Development board](#) section. If you wish to choose some other resources, make sure that the third party debugger scripts also configure the same resources.

5 PSoC™ 6 application notes

DRAFT

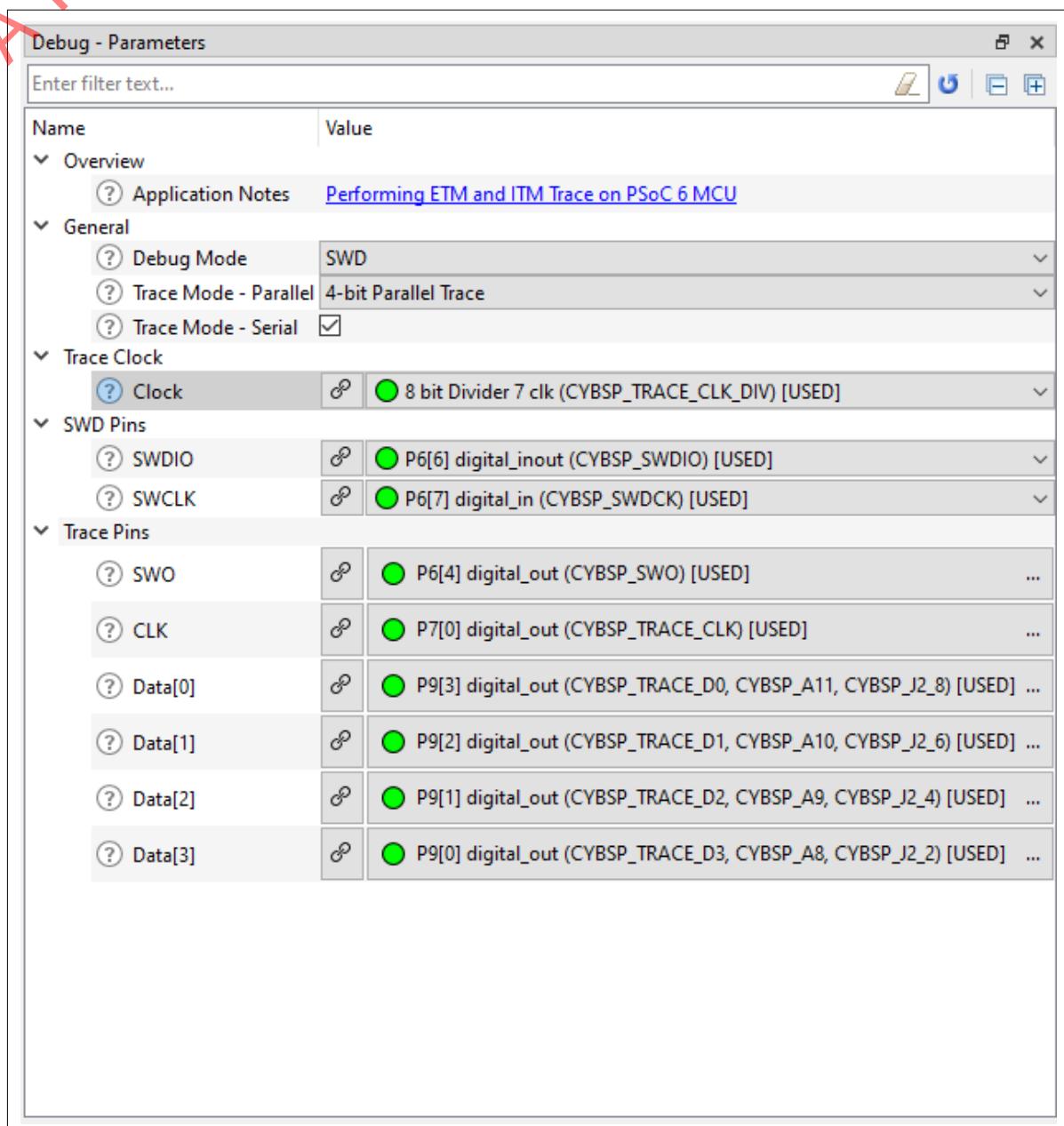


Figure 605 CY8CEVAL-062S2 kit trace pin schematics

- Click **File > Save**. Close the Device Configurator.

For detailed steps on creating a new project in Eclipse IDE for ModusToolbox™, see [AN228571 - Getting started with PSoC™ 6 MCU on ModusToolbox™ software](#).

5.18.4.2 Performing trace on IAR Embedded Workbench

5.18.4.2.1 Import the ModusToolbox™ project into IAR EW

To export the project from ModusToolbox™ to IAR EW, do the following:

5 PSoC™ 6 application notes

- ~~DRAFT~~
1. Open **modus-shell** and navigate to the **Empty App** application directory.
 2. Run the following command:

```
make ewarm8 TOOLCHAIN=IAR
```

To import the project into IAR EW, do the following:

1. Start IAR Embedded Workbench.
2. On the main menu, select **Project > Create New Project > Empty project** and click **OK**.
3. Browse to the **Empty App** application directory, enter a desired application name, and click **Save**.
4. After the application is created, select **File > Save Workspace**. Then, enter a desired workspace name.
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**.
- The project will be created in the workspace window.
7. Right-click the project and click **Make** to build it.

5.18.4.2.2 Configure the debugger script and debugger

The debugger configures the TPIU port and necessary clocks before starting the trace. This configuration is done through functions listed in a debugger script. The debugger scripts for PSoC™ 6 MCUs are located in the IAR EW installation directory: <IAR Installation path> \Embedded Workbench 8.4\arm\config\debugger\Infineon\PSoC6. The debugger scripts have the .dmac extension.

The debugger scripts have default values for the TPIU port and clock divider selection. The default value for the TPIU port is usually not applicable for all development kits. Therefore, the script file must be edited to choose the correct TPIU port.

The CY8CEVAL-062S2 evaluation kit uses the CY8C62xA PSoC™ 6 MCU device. Open the **CY8C6xxA_CM4.dmac** debugger script from the IAR EW installation directory (<IAR Installation path> \Embedded Workbench 8.4\arm\config\debugger\Infineon\PSoC6). In the file, search for the **_TRACE_path** parameter.

The **_TRACE_path** parameter in the script determines the GPIO port to configure before starting trace. The parameter and its comment are as shown below:

```
// The parameter could be any combination of TraceDx routes:  
// 0x0000 - All Trace Data pins routed to port 7  
// 0x1111 - All Trace Data pins routed to port 9  
// 0x2222 - All Trace Data pins routed to port 10  
// 0x0001 - TraceD3..TraceD1 pins routed to port 7 and TraceD0  
//           routed to port 9. Configuration for CY8CKIT-062-BLE board. (default)  
//  
__param _TRACE_path = 0x0001;
```

Figure 606 IAR debugger script snapshot for PSoC™ 6 MCU

As seen in the [Development board](#) section, the trace data pins are routed to **Port 9** of the kit. Therefore, the default value of **0x0001** for the **_TRACE_path** parameter is not correct in this case and must be modified.

Do the following to override the parameters values in the debugger script without editing the file directly:

~~5 PSoC™ 6 application notes~~

1. In the IAR EW, go to **Project > Options > Debugger > Extra Options** and select **Use command line options**.
2. In the text box, enter the following:

```
macro_param _TRACE_path=0x1111
```

Note that this command is space-sensitive; extra spaces shouldn't be added around the equals sign.

You can modify any other parameter in the script in a similar way if required. For the flow followed in this application note, modifying the trace path is sufficient.

To select the debugger, in the IAR EW, go to **Project > Options > Debugger > Setup** and select **I-jet** under the **Driver** drop-down.

5.18.4.2.3 Perform ETM trace

In the `main.c` file, modify the main function as shown below. This will toggle the user LED on the kit every second.

Code Listing 1

```
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 User LED */
    result = cyhal_gpio_init(CYBSP_USER_LED, CYHAL_GPIO_DIR_OUTPUT,
                            CYHAL_GPIO_DRIVE_STRONG, CYBSP_LED_STATE_OFF);

    /* GPIO init failed. Stop program execution */
    if (result != CY_RSLT_SUCCESS)
    {
        CY_ASSERT(0);
    }
    for (;;)
    {
        cyhal_system_delay_ms(1000);

        /* Invert the USER LED state */
        cyhal_gpio_toggle(CYBSP_USER_LED);
    }
}
```

To start the ETM trace, do the following:

1. Click the **Download and Debug** icon on the IAR EW toolbar to program the image and start the debug perspective.

5 PSoC™ 6 application notes

- ~~2.~~ Select **I-jet > ETM trace**. The ETM trace window appears.
- ~~3.~~ Right-click the ETM trace window and click **Enable**.
- ~~4.~~ Click the **Go** icon on the toolbar to start the execution and collect the ETM trace data.

You should see the user LED toggling every second. The following is a snapshot from the debug log window:

```

Debug Log
Log
Tue May 31, 2022 10:48:46: Trace/TPIU: Initializing in 4-bit mode ...
Tue May 31, 2022 10:48:46: Trace/TPIU: Initialization DONE
Tue May 31, 2022 10:48:47: Trace: Calibration [4-bitclk=4.02MHz,auto]: OK=#0, pattern=+++++*****C+++++*****+
Tue May 31, 2022 10:48:47: Trace/Streaming: Active.
Tue May 31, 2022 10:48:47: Trace/Streaming/Done: nInst=6629
Tue May 31, 2022 10:48:47: Trace/Streaming/Total: nInstHit=6629 nCovFull=647 nCovPart=38
Tue May 31, 2022 10:48:47: Trace: ETM read and validated (2016B in 27ms - total 2016B kept)
Tue May 31, 2022 10:53:57: Trace/Streaming: Active.
Tue May 31, 2022 10:53:57: Trace/Streaming: ETM clock change detected - streaming stopped to avoid unstable capture.
Tue May 31, 2022 10:53:57: Trace/Streaming/Done:
⚠️ Tue May 31, 2022 10:53:57: Warning: Trace streaming aborted because ETM clock change during capture was detected.

```

Figure 607 IAR debug log for ETM trace abortion warning.

The trace calibration detects a clock of 4 MHz before entering the main function. This is because the debug scripts configure only the clock divider and not the FLLs/PLLs that generate the high-frequency peripheral clock from the internal main oscillator (**IMO – 8MHz**). Therefore, when the code just starts executing from the beginning of the main function, the FLLs/PLLs are disabled and the IMO is routed directly to the high-frequency clock. The debugger will by default apply a **divide by 2** value on the clock, giving **4 MHz**.

Once the code executes the `cybsp_init()` function, the FLLs/PLLs are enabled and the high-frequency peripheral clock will now have a value of **100 MHz**. The debugger will now see a different clock and stops trace. The logs highlighted in the previous **Debug log** window reports the same.

The ideal way to trace is to put a breakpoint right after the `cybsp_init()` function. Once the breakpoint is hit, trace stops. Now if you restart the trace, the debugger will recalibrate to the new frequency and start capturing the data correctly. See the following snapshot from the debug log window:

```

Debug Log
Log
Tue May 31, 2022 11:12:52: Trace: ETM read but aborted (6048B in 35ms - total 8.2KB kept)
Tue May 31, 2022 11:12:52: Trace/Streaming: ETM clock change detected - streaming stopped to avoid unstable capture.
Tue May 31, 2022 11:12:52: Trace/Streaming/Done:
⚠️ Tue May 31, 2022 11:12:52: Warning: Trace streaming aborted because ETM clock change during capture was detected.
Tue May 31, 2022 11:12:52: Breakpoint hit: Code @ main.c:25.5, type: default (auto)
Tue May 31, 2022 11:12:53: Trace: Calibration [4-bitclk=50.23MHz,auto]: OK=#19, pattern=+++++*****X*+*****C+++++*****+
Tue May 31, 2022 11:12:53: Trace/Streaming: Active.

```

Figure 608 IAR debug log for ETM trace recalibration

If you pause the trace, the trace data collected will be populated in the **ETM Trace** window as shown below:

Timestamp	Address	Exec	Trace	Exc...	Access	Data Address	Data Value	Comment
			ADD\$ r0, r0, #1 Cy_DelayCycles_loop:					
894496626	0x100022e6	Thumb	ADD\$ R0, R0, #1 SUB\$ r0, r0, #2					
894496627	0x100022e8	Thumb	SUB\$ R0, R0, #2 BNE Cy_DelayCycles_loop					
894496628	0x100022ea	Thumb	BNE.N Cy_DelayCycles_loop ... ADD\$ r0, r0, #1					
894496630	0x100022e6	Thumb	Cy_DelayCycles_loop: ADD\$ R0, R0, #1					

Figure 609 IAR ETM trace window showing ETM data

5 PSoC™ 6 application notes

5.18.4.2.4 Perform ITM trace (printf-style debugging)

1. In IAR EW, go to **Project > Options > I-jet > Trace** and select **Serial (SWO)** under the **Mode** drop-down list. Uncheck the **Allow ETB** checkbox.
2. Go to **Project > Options > General Options > Library Configuration**. Select **Semihosted, Via SWO** and select the **Use CMSIS** checkbox.

In the `main.c` file, modify the main function and header list to include the `printf` statement as shown below:

Code Listing 2

```
#include "cy_pdl.h"
#include "cyhal.h"
#include "cybsp.h"
#include "stdio.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 User LED */
    result = cyhal_gpio_init(CYBSP_USER_LED, CYHAL_GPIO_DIR_OUTPUT,
                            CYHAL_GPIO_DRIVE_STRONG, CYBSP_LED_STATE_OFF);

    /* GPIO init failed. Stop program execution */
    if (result != CY_RSLT_SUCCESS)
    {
        CY_ASSERT(0);
    }

    int i = 0;

    for (;;)
    {
        cyhal_system_delay_ms(1000);

        /* Invert the USER LED state */
        cyhal_gpio_toggle(CYBSP_USER_LED);

        printf("Hello empty project - %d\n", i++);
    }
}
```

To start the trace, do the following:

~~5 PSoC™ 6 application notes~~

- ~~1.~~ Click the **Download and Debug** icon on the IAR EW toolbar to program the image and start the debug perspective.
- ~~2.~~ Go to **I-jet > SWO Configuration**. Make the ITM Stimulus Ports configuration as follows:

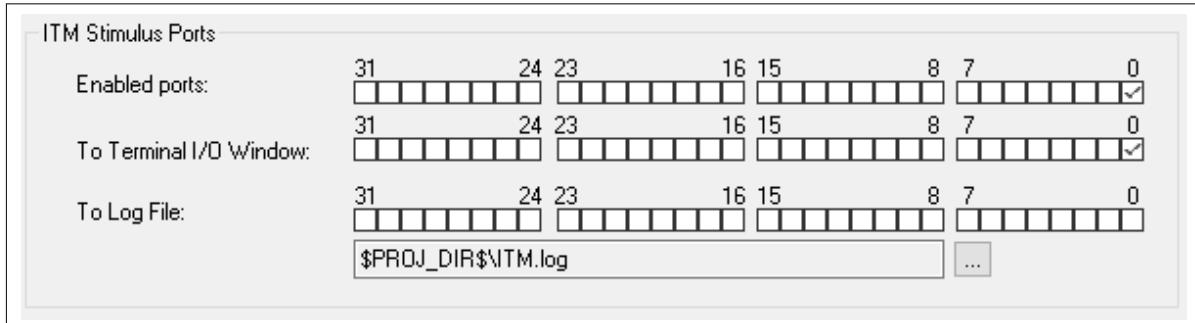


Figure 610 IAR SWO configuration for ITM

- ~~3.~~ Go to **View tab** and select **Terminal I/O**. A terminal I/O window will open; this is where you will see the print messages.
- ~~4.~~ Click the **Go** icon on the toolbar to start the execution and collect **ITM data**.

You should see the user LED toggling every second. The Terminal I/O window should display logs as shown below:

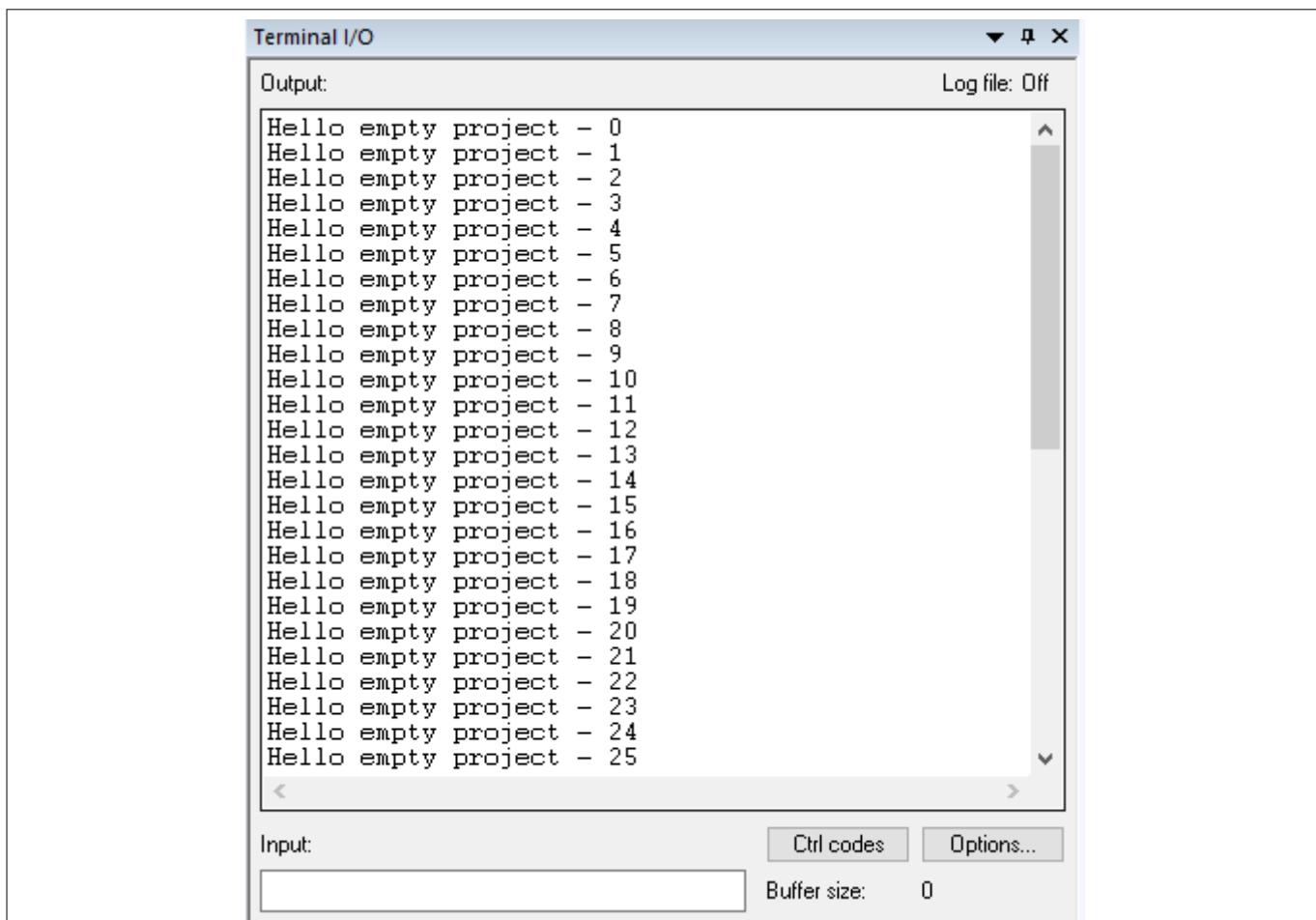


Figure 611 IAR terminal i/o window showing printf output

You can also call the `scanf` function to read from the Terminal I/O.

~~5 PSoC™ 6 application notes~~

5.18.4.3 Performing trace on Keil µVision

5.18.4.3.1 Import the ModusToolbox™ project into Keil µVision

To export the project from ModusToolbox™ to Keil µVision, do the following:

1. Open **modus-shell** and navigate to the Empty App application directory.
2. Run the following command:

```
make uvision5 TOOLCHAIN=ARM
```

To import the project into µVision, do the following:

1. Open the Empty App directory in file explorer.
 2. Double-click the *.cprj file. This launches the Keil µVision IDE. The first time you do this, a pack installer dialog box will appear.
 3. Follow the instructions on the screen to install the **Infineon::PSoC6_DFP** pack (v1.3 or higher). This pack can also be installed separately by launching the Pack Installer tool in µVision. The project will be created in the project window.
 4. Right-click the project and select **Options for Target ‘<application-name>**.
 5. Select the **Device** tab. Depending on the kit you are using, select the relevant PSoC™ 6 MCU device under Infineon list.
- For the CY8CEVAL-062S2 kit, select **Infineon > PSoC™ 62 > CY8C62xA > CY8C624ABZI-S2D44 > CY8C624ABZI-S2D44:Cortex-M0p**. Click **OK**.
6. Repeat the previous step and select **Infineon > PSoC™ 62 > CY8C62xA > CY8C624ABZI-S2D44 > CY8C624ABZI-S2D44:Cortex-M4**. Click **OK**.
 7. At this point, you should see a folder named “DebugConfig” created in your Empty App project directory. The DBGCONF files in this folder contains the TPIU port selection information.
 8. Open the project options again.
 9. Select the **C/C++ (AC6)** tab and do the following and click **OK**:
 - Set **Language C** to **c99**.
 - Set **Warnings** to **AC5-like Warnings**.
 - Set **Optimizations** to **-Os balanced**.
 10. Right-click the project and click **Build Target**.

5.18.4.3.2 Configure the debugger script and debugger

The debugger configures the TPIU port and necessary clocks before starting the trace. The PSoC6_DFP pack has the necessary files to perform this configuration. As explained earlier, the DBGCONF files in the DebugConfig folder located under the project directory has the TPIU port selection information. The default value for the TPIU port is usually not applicable for all development kits; therefore, the DBGCONF file must be edited to choose the correct TPIU port.

As seen in the [Development board](#) section, the trace data pins are routed to Port 9 of the kit.

Do the following to override the default values in the DBGCONF file:

1. In µVision, select **File > Open**. Navigate to the DebugConfig folder in the Empty App project directory, select the CM4 DBGCONF file, and click **Open**.
2. On the editor that opens with the file content, at the bottom of the editor, select **Configuration Wizard**.

5 PSoC™ 6 application notes

DRAFT

```
// File: CY8C6xxA.dbgconf
// Version: 1.0
// <<< Use Configuration Wizard in Context Menu >>>
// <h>Trace Pin Setup
// <o> TPIU Pin Location
//   <0=> Pin Location 0 (Do not configure)
//   <1=> Pin Location 1 (CY8CKIT-062-WIFI-BT Pioneer Kit like)
//   <2=> Pin Location 2 (CY8CKIT-062S2-43012 Pioneer Kit like)
// <i> Select TPIU pin location for your board configuration:
// <i> - Pin Location 0 (No one pin will be configured for trace purpose)
// <i> - Pin Location 1 (TRACECLK: P7_0, TRACEDATA0: P9_3, TRACEDATA1: P7_6, TRACEDATA2: P7_5, TRACEDATA3: P7_4)
// <i> - Pin Location 2 (TRACECLK: P7_0, TRACEDATA0: P9_3, TRACEDATA1: P9_2, TRACEDATA2: P9_1, TRACEDATA3: P9_0)
// <i> Default: Pin Location 0
__TPIU_pinlocation = 2;

// </h>
// <<< end of configuration section >>>
```

Figure 612 Opening the DBGCONF file in Configuration Wizard

- Select **Trace Pin Setup > TPIU Pin Location > Pin Location 2** from the drop-down list. Details about the drop-down options are given at the end of the editor. Close the file once the selection is complete.

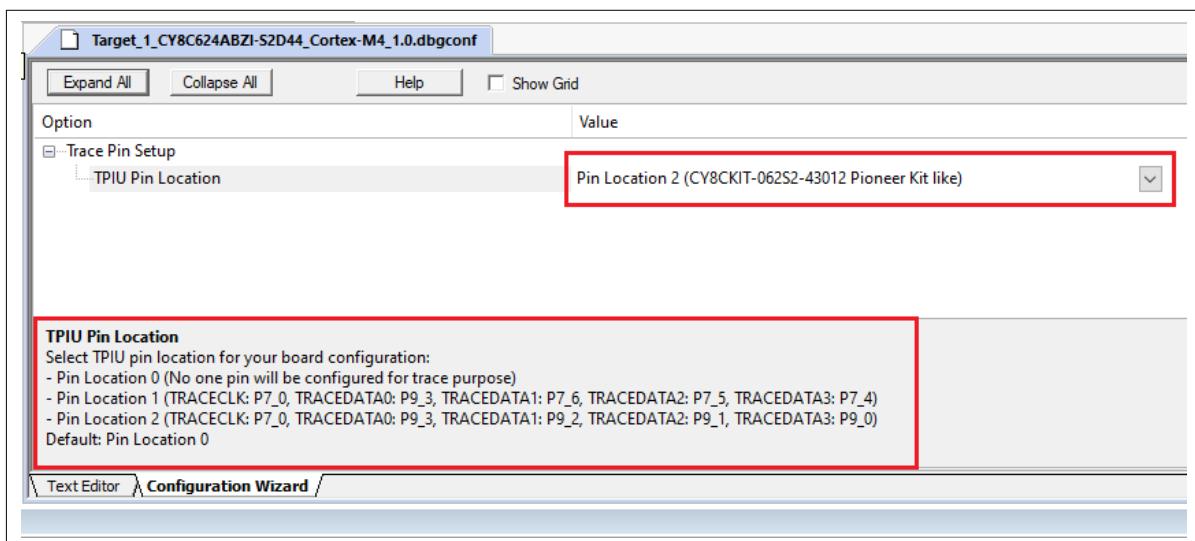


Figure 613 Editing the DBGCONF file in Configuration Wizard

To select and configure the debugger in μVision, do the following:

- Open project options.
- Select **Debug** tab. Select the debugger as **ULINK Pro Cortex Debugger** from the drop-down menu. Click the **Setting** button next to it.

5 PSoC™ 6 application notes

DRAFT

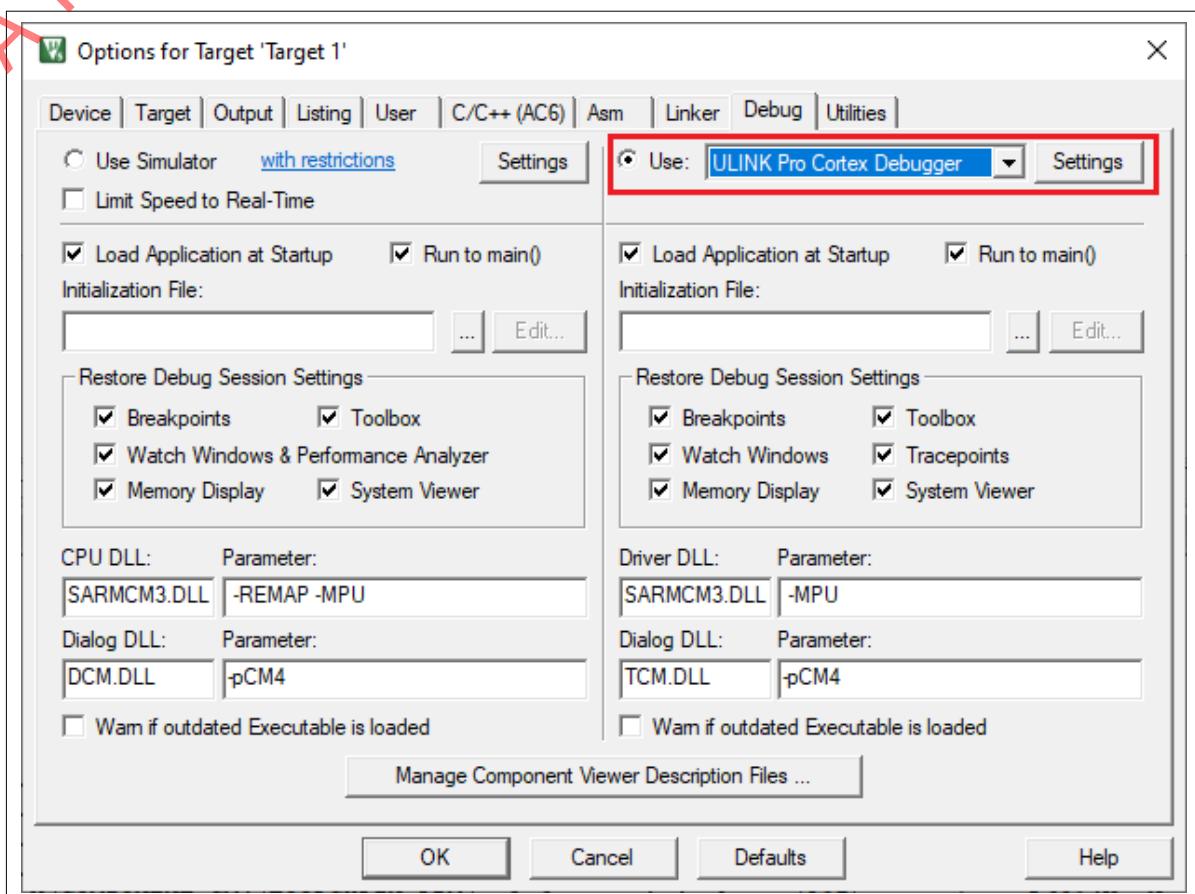


Figure 614 Configurations in Debug tab

3. In the new window, match the settings of the **Debug** tab as shown in the following image:

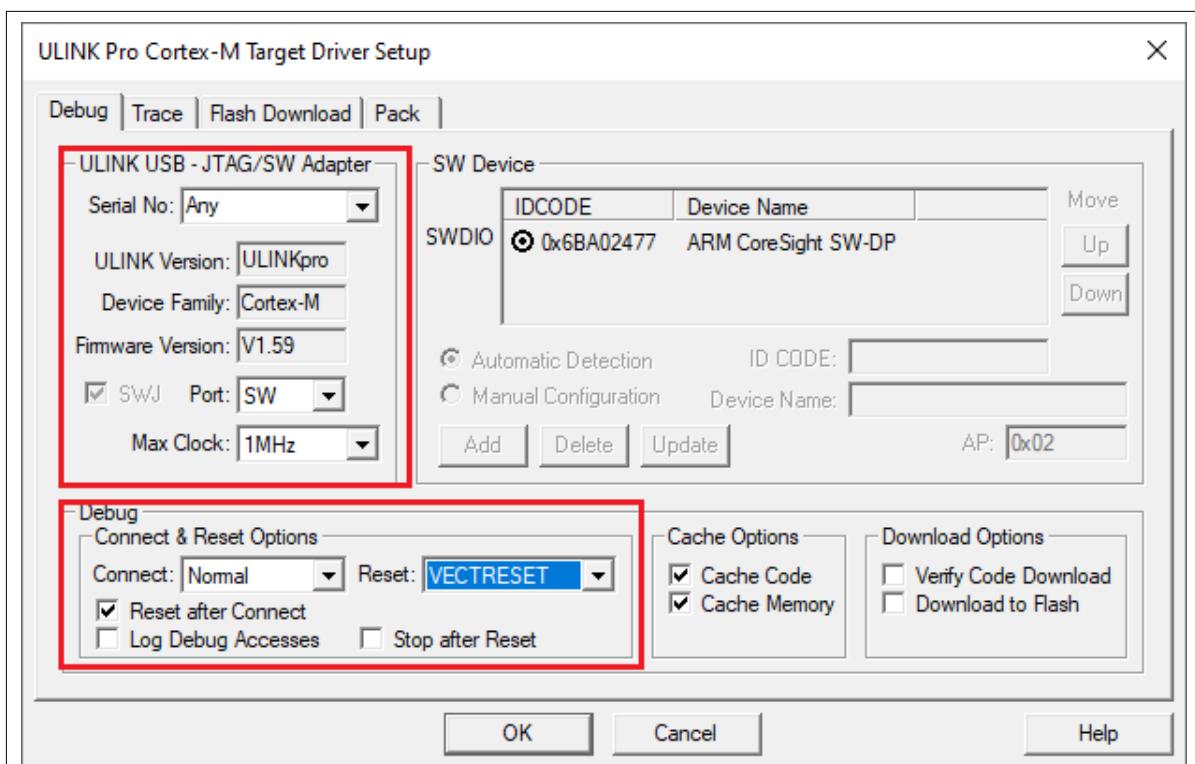


Figure 615 Configurations in Debug > Debug tab

5 PSoC™ 6 application notes

- DRAFT**
- Select the **Trace** tab and match the setting as shown in the following image:

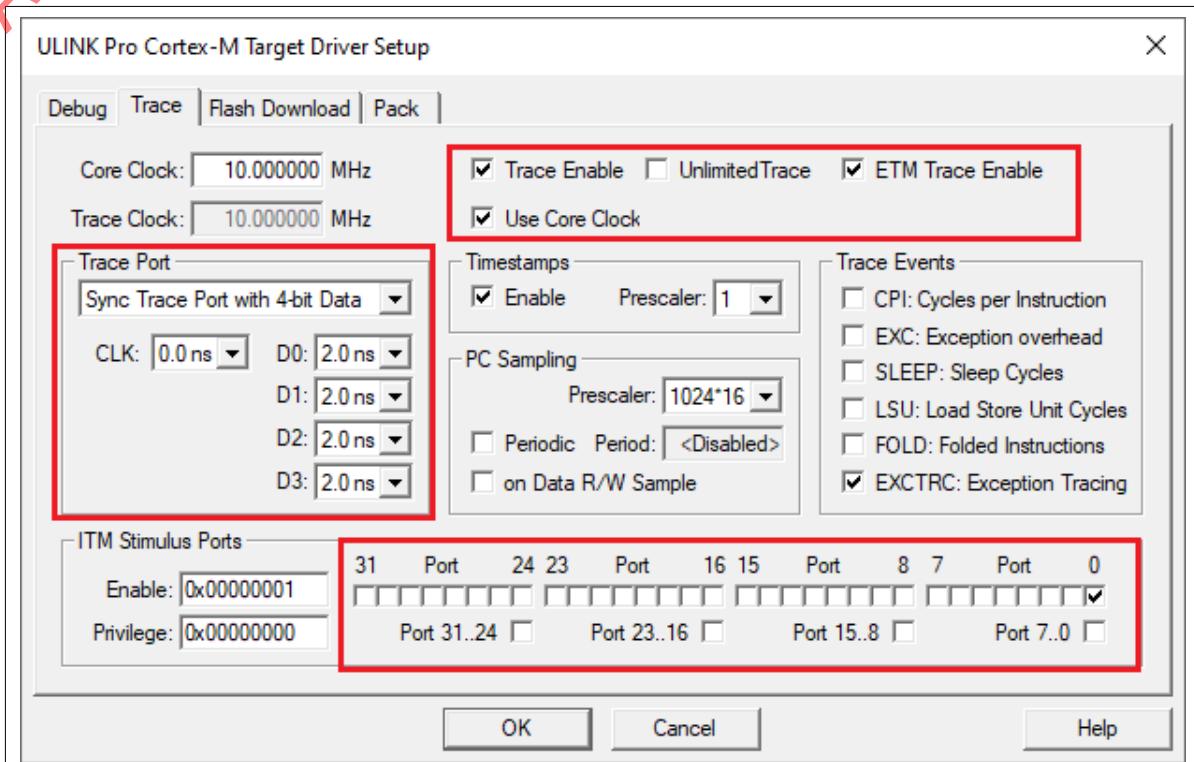


Figure 616 Configurations in Debug > Trace tab

- Select the **Flash Download** tab, select the check boxes for **Program**, **Verify**, and **Reset and Run**.
- Select the **Pack** tab, and select the **Enable** check box. Click **OK**.

5.18.4.3.3 Perform ETM trace

In the `main.c` file, modify the code as per [Code Listing 1](#) in the [Perform ETM trace](#) section. This will toggle the user LED on the kit every second.

To start ETM trace, do the following:

- Click the **Build** icon on the μ Vision toolbar to rebuild the code.
- Click the **Start/Stop Debug Session** icon on the μ Vision toolbar to program the image and start the debug perspective.
- Select **View > Trace > Trace Data** to open an ETM trace window.
- Click the **Run** icon on the toolbar to start the execution and collect the ETM trace data.

You should see the user LED toggling every second. If you pause the trace, the trace data collected will be populated in the ETM trace data window as shown below:

Trace Data					
Time	Address / Port	Instruction / Data	Src Code / Trigger Addr	Function	
X : 0x10005286	ADDS r0,r0,#1		ADDS r0, r0, #1 ; 1 2 In...	Cy_SysLib_DelayCycles	
X : 0x10005288	SUBS r0,r0,#2		SUBS r0, r0, #2 ; 1 2 De...	Cy_SysLib_DelayCycles	
X : 0x1000528A	BNE 0x10005286		BNE Cy_DelayCycles_loop ; (1)...	Cy_SysLib_DelayCycles	
X : 0x10005286	ADDS r0,r0,#1		ADDS r0, r0, #1 ; 1 2 In...	Cy_SysLib_DelayCycles	
X : 0x10005288	SUBS r0,r0,#2		SUBS r0, r0, #2 ; 1 2 De...	Cy_SysLib_DelayCycles	
2,818,378,959,600 s	BNE 0x1000528A		BNE Cy_DelayCycles_loop ; (1)...	Cy_SysLib_DelayCycles	

Figure 617 μ Vision trace data windows showing the ETM data

5 PSoC™ 6 application notes

The ULINKpro debugger will automatically calibrate to the new trace clock if the trace clock changes during code execution.

5.18.4.3.4 Perform ITM trace (printf-style debugging)

To configure for printf-style debugging, do the following:

1. In μVision, launch the **Manage Run-Time Environment** window.
2. Select **Compiler > IO**, and then select the check boxes for **STDIN** and **STDOUT**. In the drop-down list for STDIN and STDOUT, select **ITM**.
3. In the `main.c` file, modify the main function and header list per [Code Listing 2](#) in the **Perform ITM trace (printf-style debugging)** section.

To start trace, do the following:

1. Click the **Build** icon on the μVision toolbar to rebuild the code.
2. Click the **Start/Stop Debug Session** icon to program the image and start the debug perspective.
3. Select **View > Serial Windows > Debug (printf) Viewer**.
4. Click the **Run** icon on the toolbar to start the execution and collect the ITM data.

You should see the user LED toggling every second. The Debug Viewer should display logs as shown below:

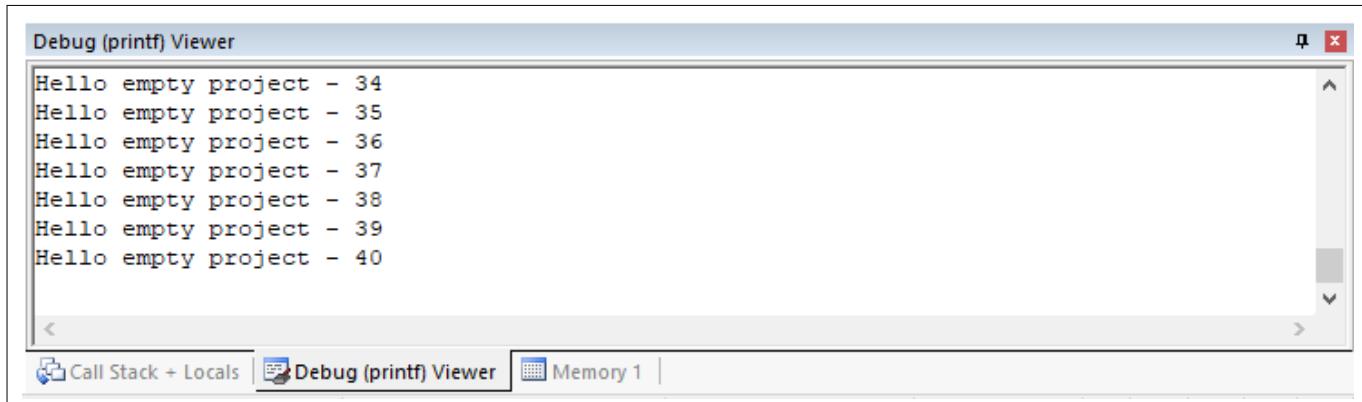


Figure 618 μVision debug viewer showing printf outputs

5 PSoC™ 6 application notes~~DRAFT~~
5.18.5 Summary

This application note explored the PSoC™ 6 MCU trace architecture and the associated development tools. It also covered various general aspects of Arm® trace architecture. With this background, you should be able to understand and perform trace on PSoC™ 6 MCU to its full potential.

Although the application note limits itself to showcasing the ETM and ITM trace, third party tools in general offer a wide variety of other trace features such as function profiling, data watchpoints, interrupt logging, and code coverage. You can refer to the respective tool documentation to understand all the features they offer and use them effectively for complex embedded application debugging and development.

5 PSoC™ 6 application notes

References

-
- 1
- 8
- [1] AN228571 - Getting started with PSoC™ 6 MCU on ModusToolbox™ software
- [2] ModusToolbox™ 3.0 user guide
- [3] Learn the architecture: Understanding trace by Arm®
- [4] IAR Embedded Workbench IDE user guide
- [5] Keil µVision user guide

~~5 PSoC™ 6 application notes~~

~~5.18.6 Revision history~~

Document version	Date of release	Description of changes
**	2022-09-22	Initial release.
*A	2022-12-09	Section 3.1.2 - Added a note and fixed errors.

5.19 AN235297 Creating a ModusToolbox™ 3.x BSP

About this document

-
- 1
- 9

Scope and purpose

This application note describes how to create a Board Support Package (BSP) using the ModusToolbox™ BSP Assistant tool. This document explains the basics of a BSP along with the various use cases where the BSP Assistant tool can be useful during application development using ModusToolbox™ version 3.0 or above. Applications developed with ModusToolbox™ version 3.0 are not backward compatible with earlier versions of ModusToolbox™.

Intended audience

This document is intended for anyone who needs to create a user-specific design board using an Infineon device supported inside the ModusToolbox™ ecosystem.

Document conventions

Convention	Explanation
Bold	Emphasizes heading levels, column headings, menus and sub-menus.
<i>Italics</i>	Denotes file names and paths.
Courier New	Denotes APIs, functions, interrupt handlers, events, data types, error handlers, file names, directories, command line inputs, code snippets, etc.
File > New	Indicates that a cascading sub-menu opens when you select a menu item.

Abbreviations and definitions

Abbreviation	Meaning
BSP	Board Support Package
MCU	Microcontroller Unit

5 PSoC™ 6 application notes

5.19.1 Introduction

5.19.1.1 What is a BSP?

BSPs are a set of files and directories that provide the necessary functionality to develop target applications on any given board. The board is typically a printed circuit board (PCB) used in any electronics product like a mobile phone, laptop, digital camera, etc. These boards usually have a microcontroller (or microprocessor) chip with various peripherals and other components that are wired together to meet the target application requirements.

Infineon has a range of microcontroller devices belonging to various families such as PSoC™ 4, PSoC™ 6, and XMC™, and provides development kits (or boards) for the evaluation of these devices. The BSPs for these development boards are made available through the ModusToolbox™ ecosystem in the [Infineon GitHub website](#).

5.19.1.2 BSP Assistant overview

The BSP Assistant tool helps you create and manage custom BSPs for the board designed for your application using Infineon MCUs. The tool is available in both graphical user interface (GUI) and command line interface (CLI) versions.

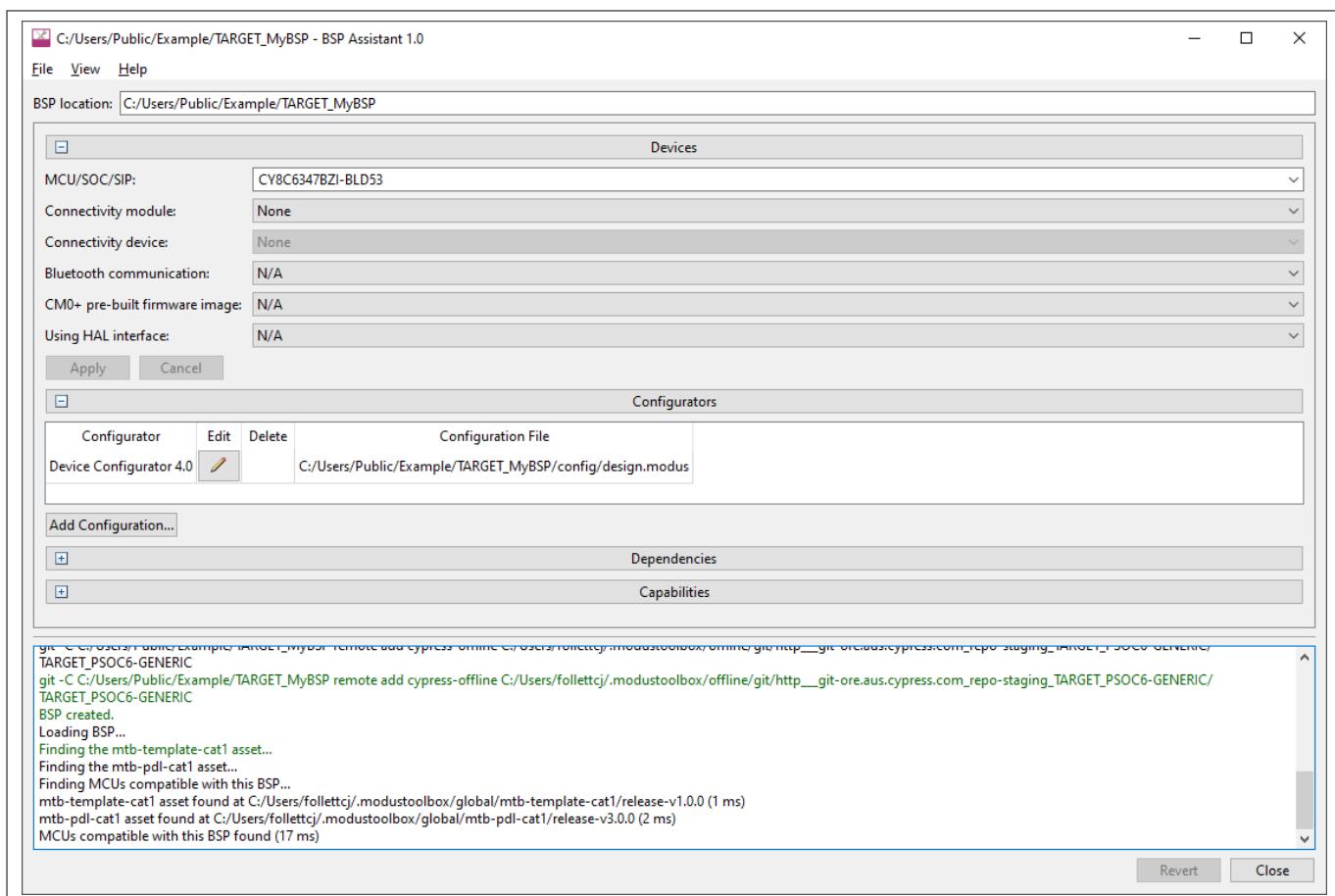


Figure 619 **BSP Assistant GUI**

5 PSoC™ 6 application notes~~DRAFT~~
5.19.1.3 Overview

This application note has the following sections to help you learn the following:

- **BSP design** - Explains the BSP architecture
- **Using the BSP Assistant tool** - Explains typical BSP Assistant tool use cases such as creating and modifying BSPs using the **Hello World** code example
- **Advanced usage** - Explains advanced BSP Assistant tool use cases like BSP migration between generations

5.19.1.4 Software requirement

Software	Minimum required version
ModusToolbox™	3.0

5 PSoC™ 6 application notes~~DRAFT~~
5.19.2 BSP design

As mentioned previously, a BSP is a set of files and directories with content specific to a target board that enables you to develop a target application. BSPs are made available through the ModusToolbox™ ecosystem via Infineon GitHub repositories.

5.19.2.1 Software

Software is provided in source or library form and contains a set of APIs to control and configure the microcontroller and other onboard components. A BSP specifies software that it requires as dependencies. For Infineon BSPs provided on GitHub, dependencies are specified in a manifest file. Once a BSP is created by the user, (either during application creation or by using the BSP Assistant) the dependencies are specified in a set of *.mtbx files in the BSP's deps subdirectory. These dependency files contain information for downloading the minimum set of libraries required to develop an application on the given board.

5.19.2.1.1 Peripheral Driver Library (PDL)

PDL contains a set of low-level APIs to control hardware peripherals like UART and SPI. The interfaces are usually specific to a particular microcontroller or microcontroller family.

5.19.2.1.2 Hardware Abstraction Layer (HAL)

HAL contains a set of high-level APIs to control hardware peripherals; the interfaces are more portable than PDL in case of the following changes:

- Changing the pin assignments for the peripherals within the same microcontroller
- Porting to another microcontroller within the same family
- Porting to another microcontroller in a different family

5.19.2.1.3 Other libraries

These libraries provide the following functionality:

- **Abstraction libraries:** These typically abstract the RTOS to help in porting the application across different RTOS like FreeRTOS or to a different board.
- **Base libraries:** These libraries, such as core-lib, core-make, recipe-make are necessary for the build process.
- **Board utilities libraries:** These are libraries supporting various utilities available on the board other than the microcontroller, such as sensors and displays, and are used to control them.
- **MCU Middleware:** These include middleware that provides RTOS services, such as FreeRTOS, or peripheral services, such as capacitive sensing.

5.19.2.2 Documentation

BSPs have the following documentation accompanying them:

- **API:** Provides details of the APIs, structures, macros, etc. that are provided as part of the BSP.
- **Board:** Provides details of the board design including that of the available microcontroller, LEDs, buttons, memory, sensors, etc.
- **README:** Provides top-level information about the BSP and usually contains links to additional documentation.
- **RELEASE:** Provides information about various versions of the BSP and changes from one version to another.

5 PSoC™ 6 application notes

5.19.2.3 Typical BSP contents

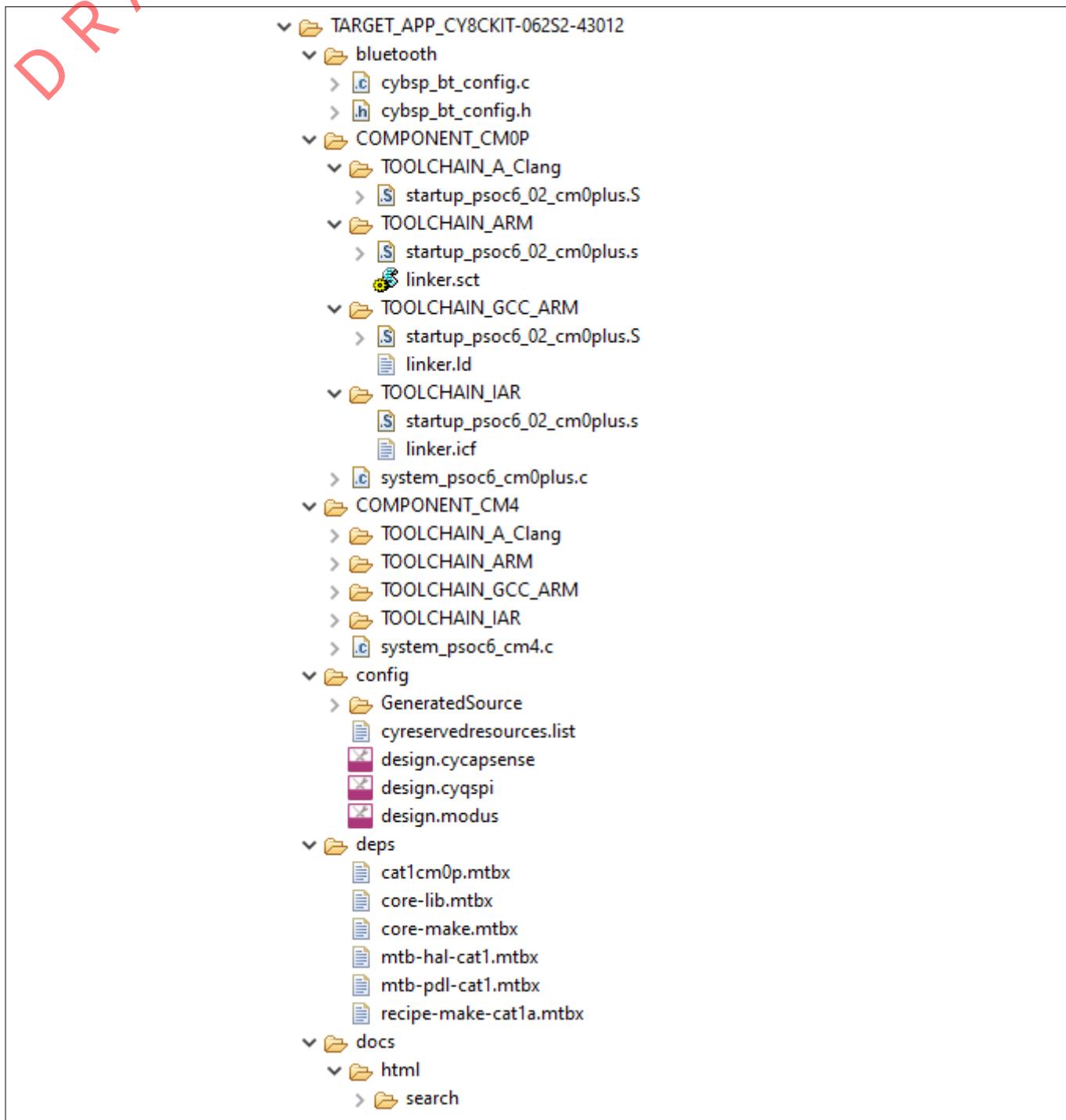


Figure 620 Typical ModusToolbox™ BSP directory structure

5.19.2.4 Startup code and linker files

Startup code is usually in assembly format and is the code that executes after a CPU reset. These are usually specific to a CPU and configure various microcontroller special-function registers such as the stack pointer.

Linker files are used to map various sections of the compiled source code into memory regions like flash and SRAM. These are used by the build system to generate the final binary that is programmed onto the target.

5 PSoC™ 6 application notes

Both startup code and linker files are specific to a build environment like GCC. BSPs that accompany the ModusToolbox™ ecosystem provides these items for the GCC, IAR, and Arm® build environments.

Since the startup code and linker files are CPU and toolchain dependent, they are located inside COMPONENT and TOOLCHAIN directories inside the BSP, which allows them to be included conditionally. For example, the directory COMPONENT_CM4/TOOLCHAIN_GCC_ARM would contain startup code and linker script files that are only used when building a project for the CM4 CPU with the GCC_ARM toolchain.

5.19.2.5 Configuration files

ModusToolbox™ comes with various BSP configurator tools that enable you to configure the microcontroller peripherals. Typically, these configure the clock, pin, and other resource-related settings in the microcontroller. There are GUI tools like the Device Configurator to open/edit the configuration files, which are saved in XML format with a specific file extension like design.modus. When the configuration is saved, the tool generates the configuration code that is linked together with the application code during the build process.

The files for each BSP configurator are located in the config directory inside the BSP.

The following table summarizes the available BSP configurator tools, associated configuration file, and a brief description.

Table 151 **BSP configuration files**

Configurator tool name	Configuration file extension	Description
capsense-configurator	*.cycapsense	CAPSENSE™ Configurator is used to create and configure CAPSENSE™ widgets, and generate code to control the application firmware.
device-configurator	*.modus	Device Configurator is used to enable and configure device peripherals, such as clocks and pins, as well as standard MCU peripherals that do not require their own tool.
qspi-configurator	*.cyqspi *.cymem	The QSPI Configurator is used to open or create configuration files, configure memory slots, and generate code for your application when external flash devices are connected to the MCU using a Quad Serial Peripheral Interface (QSPI).
seglcd-configurator	*.cyseglcd	SegLCD Configurator is used to generate display structures for the SegLCD Driver.
smartio-configurator	*.modus	Smart I/O Configurator is used to configure the smart I/O pins in the MCU.

On the **BSP Assistant** tool, do the following:

1. Click the **Edit** icon next to an existing configurator to change its settings.
2. Click **Add Configuration** to add files for a new configurator and make any necessary changes.

5 PSoC™ 6 application notes

DRAFT

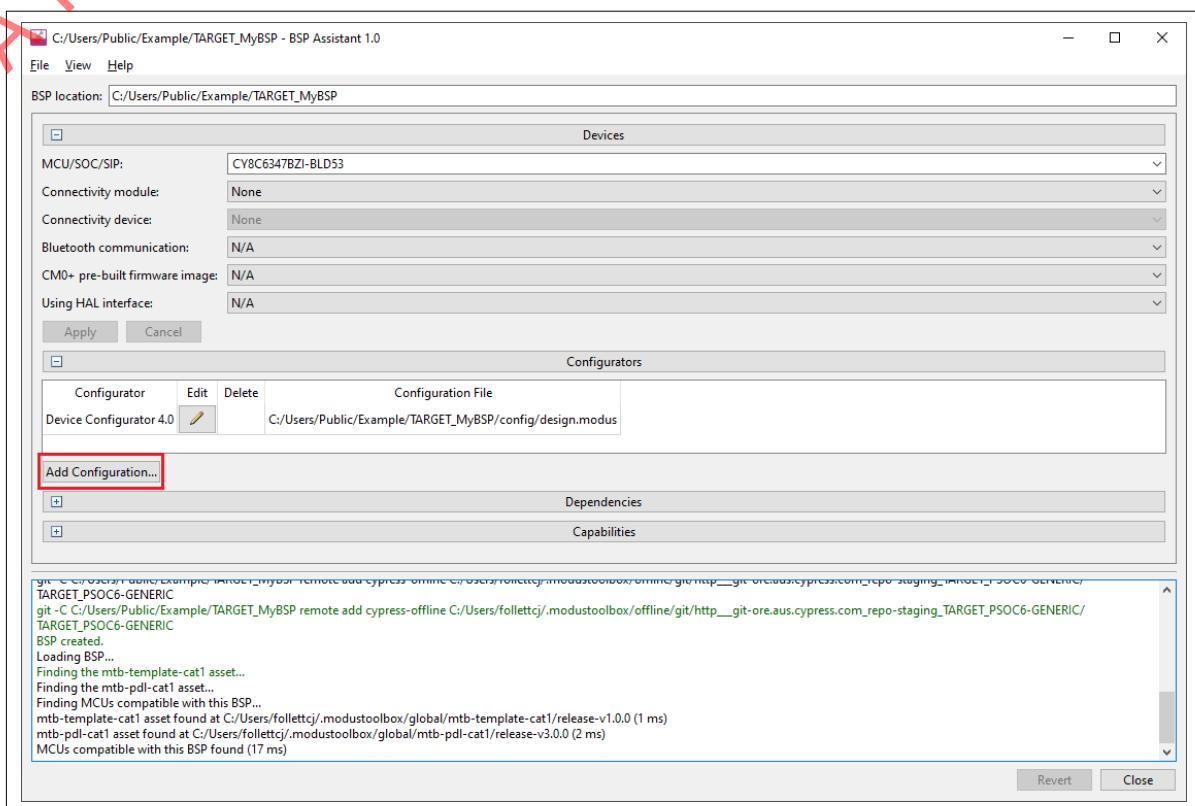


Figure 621 Add Configuration button

5.19.2.6 Generated source files

BSP Configurators generate related source/header files in the `GeneratedSource` subdirectory that are then included as part of the application build. This helps avoid writing lengthy configuration code.

5 PSoC™ 6 application notes

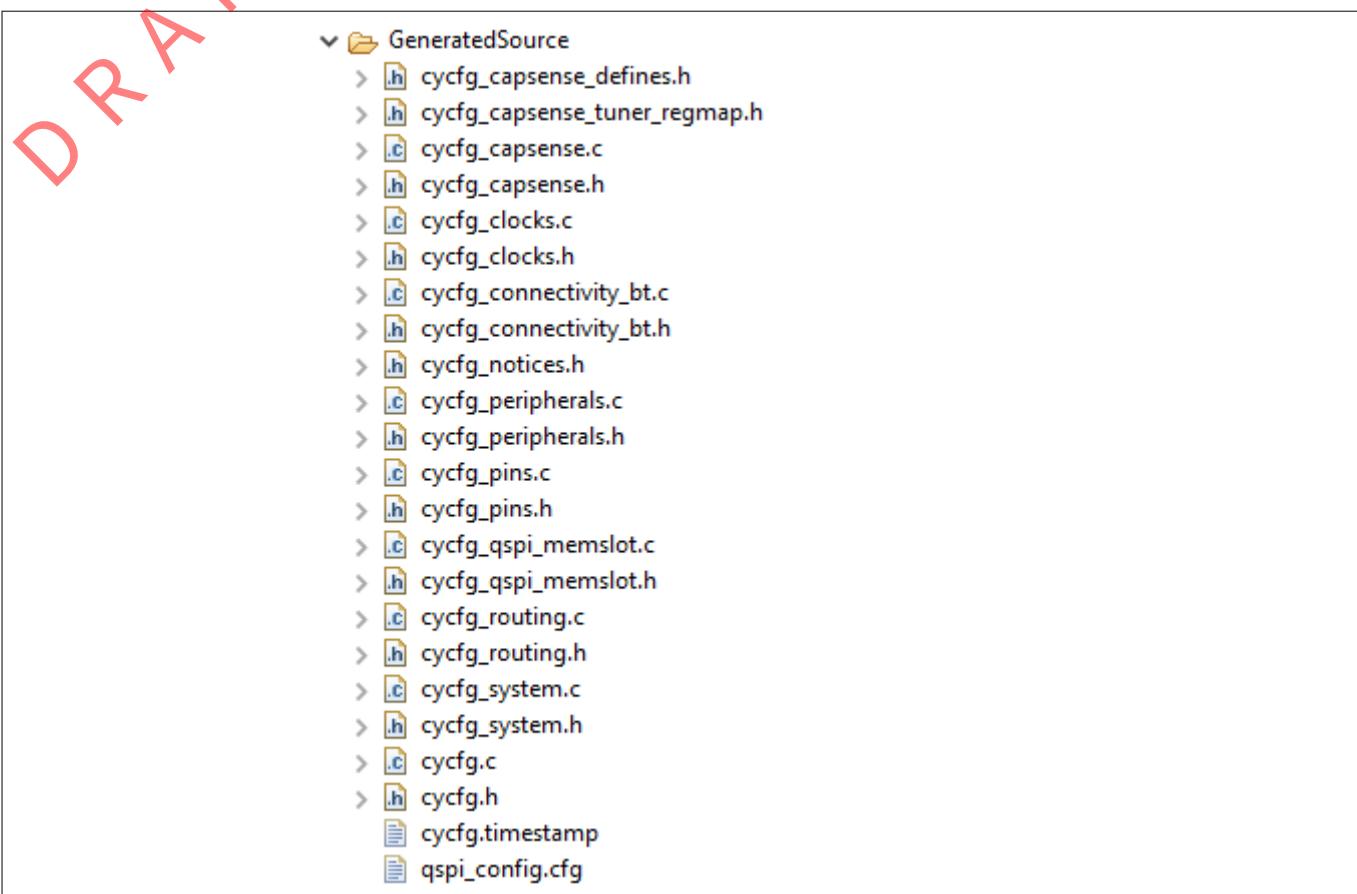


Figure 622 Sample GeneratedSource subdirectory

5.19.2.7 Static source files

A BSP contains static source files with initialization routines for the board. These must be called by the application code before using any MCU peripherals. Typical BSP static source files that are included with ModusToolbox™ BSPs are as follows:

- cybsp.c - Provides initialization code for starting up the hardware contained on the Infineon board
- cybsp.h - API header file for cybsp.c
- cybsp_doc.h - Contains code for generating BSP html documentation
- cybsp_types.h - Contains code for the states of button/pin/led on the Infineon board
- bluetooth/cybsp_bt_config.c - Provides initialization settings for the Bluetooth® module
- bluetooth/cybsp_bt_config.h - API header file for cybsp_bt_config.c

5.19.2.8 Documentation files

A BSP contains Doxygen/markup-based documentation for the application developer with the details of various libraries, release notes, etc.

~~5 PSoC™ 6 application notes~~

~~5.19.3~~ Using the BSP Assistant tool

This section describes two basic use cases for working with the BSP Assistant tool:

- [Creating a new BSP](#)
- [Customizing an existing BSP](#)

These use cases show the tools in a Windows operating system, as well as using the Eclipse IDE. If you use another IDE/OS combination, the steps will be similar but may not be identical.

5.19.3.1 Creating a new BSP

Use this workflow when you create a new board using an MCU that is not part of any of the BSPs included with the ModusToolbox™ ecosystem. For demonstration, this workflow considers creating a BSP for a board based on the [CYB06447BZI-D54](#) MCU device.

5.19.3.1.1 Create and configure the BSP

1. Type **bsp-assistant** in the Windows search tool to open the BSP Assistant tool or look in the Window's menu under "ModusToolbox <version>".
2. After the tool loads, select **File > New** to open the **Create New BSP** dialog.

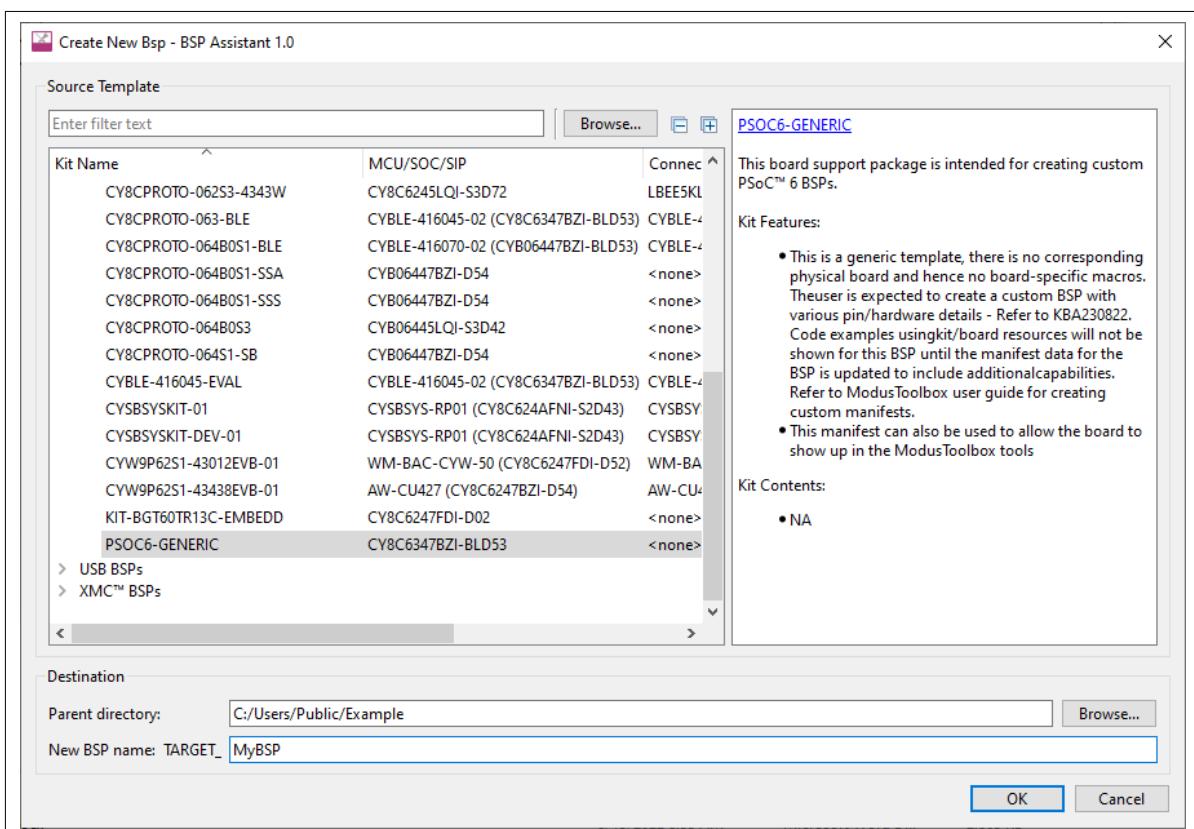


Figure 623 Create New BSP dialog

3. Select **PSOC6-GENERIC** under the **PSoC™ 6 BSPs** category or if the board you are creating is similar to an existing Infineon board, you can select it as the starting BSP.
4. Set the directory path where the BSP will be created in the **Parent Directory** text box.
5. Type a suitable name for the BSP in the **New BSP name: TARGET_** text box. This example uses the name **MyBSP**.
6. Click **OK**.

5 PSoC™ 6 application notes

The BSP Assistant tool starts downloading the contents from the Infineon [GitHub](#) website; when finished, the screen should look like the following:

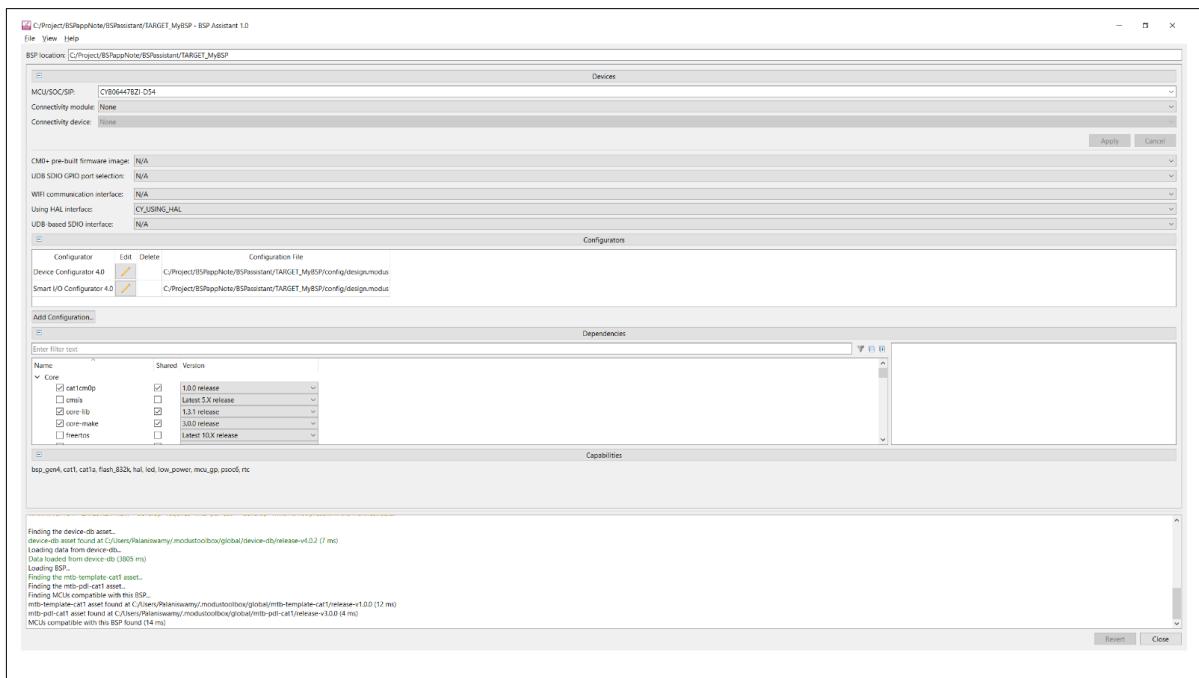


Figure 624 BSP Assistant finished loading content

7. Change the MCU device to **CYB06447BZI-D54** by selecting it from the **MCU/SOC/SIP** drop-down menu under the **Devices** section as shown and click **Apply**.

Note: You can start typing the MCU name in the drop-down box to filter the choices.

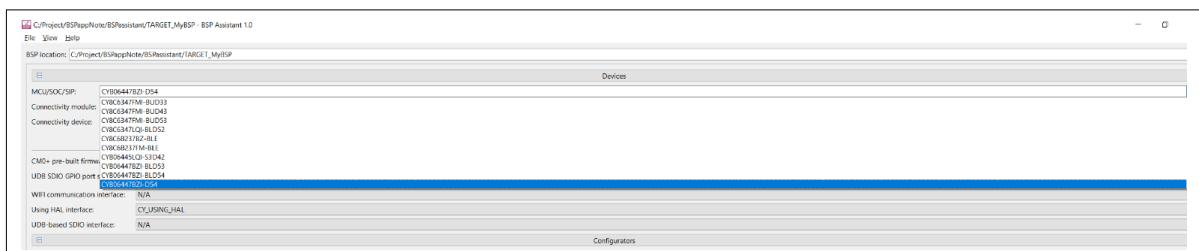


Figure 625 Changing MCU device

Messages appear in the output console ending with "Devices changed." as follows:

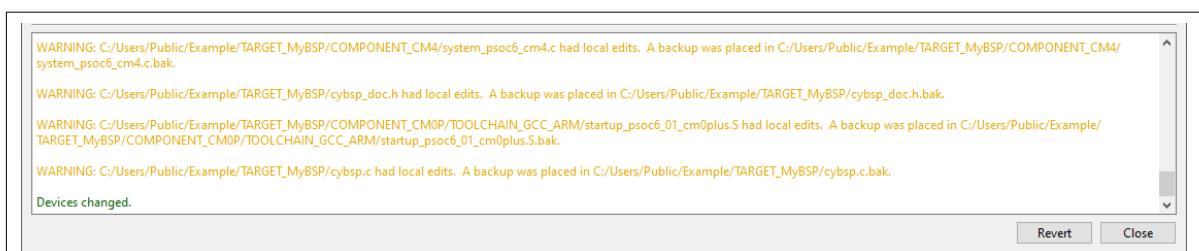


Figure 626 Output console messages

8. Under the **Configurators** section, click the **Edit** icon next to Device Configurator 4.0 to open the Device Configurator tool.

5 PSoC™ 6 application notes

DRAFT

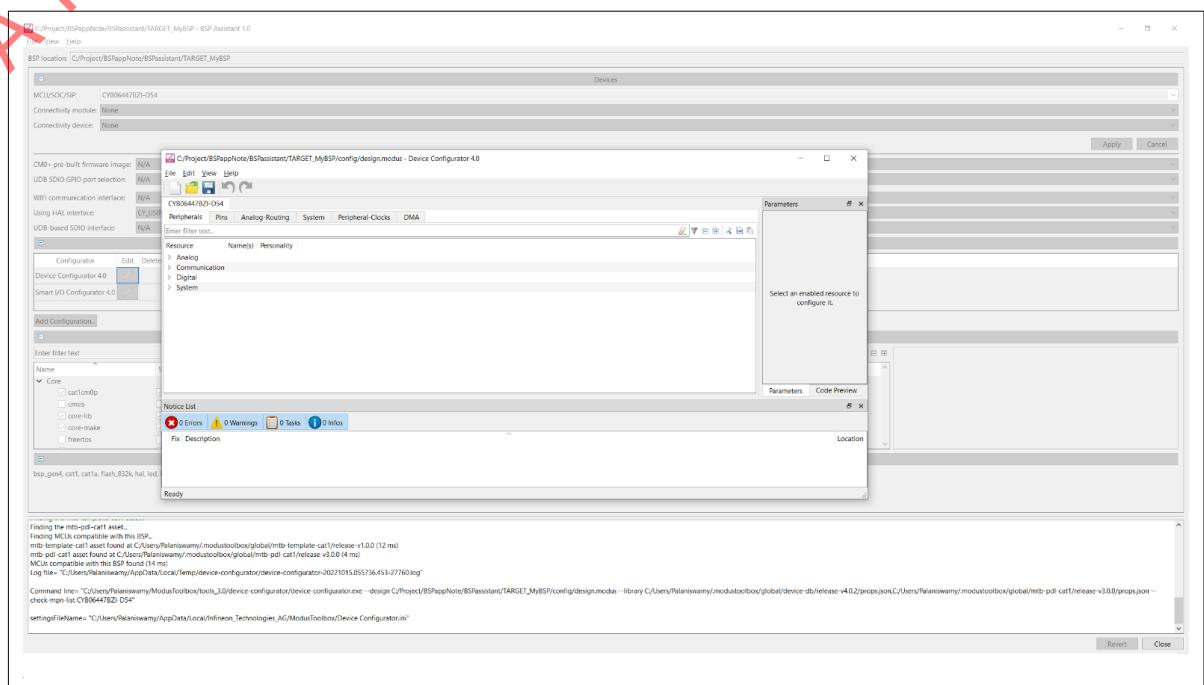


Figure 627 Opening Configurator from BSP Assistant

9. For this demo, make the following changes:

- Name the pin *P5[0]* as **CYBSP_DEBUG_UART_RX** from the **Pins** tab.
- Name the pin *P5[1]* as **CYBSP_DEBUG_UART_TX** from the **Pins** tab.
- Name the pin *P13[7]* as **CYBSP_USER_LED** from the **Pins** tab.
- Disable **CLK_ALT_SYS_TICK** from the **System** tab, **System Clocks > Miscellaneous**.

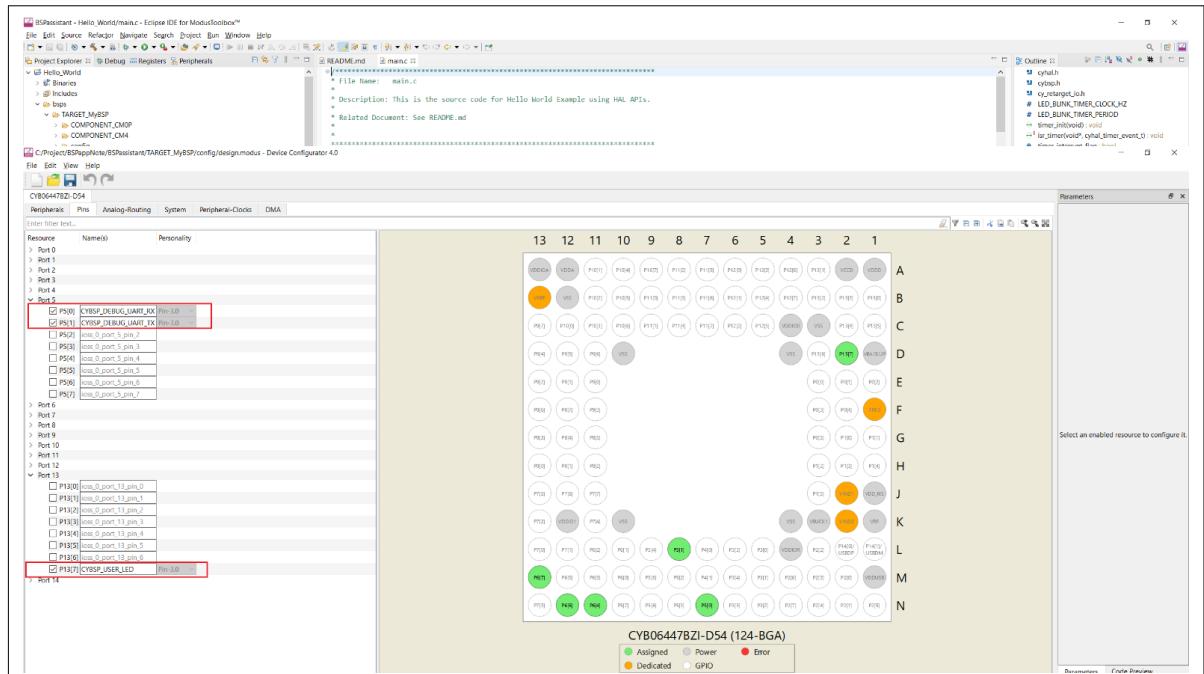


Figure 628 Changing pin settings

5 PSoC™ 6 application notes

DRAFT

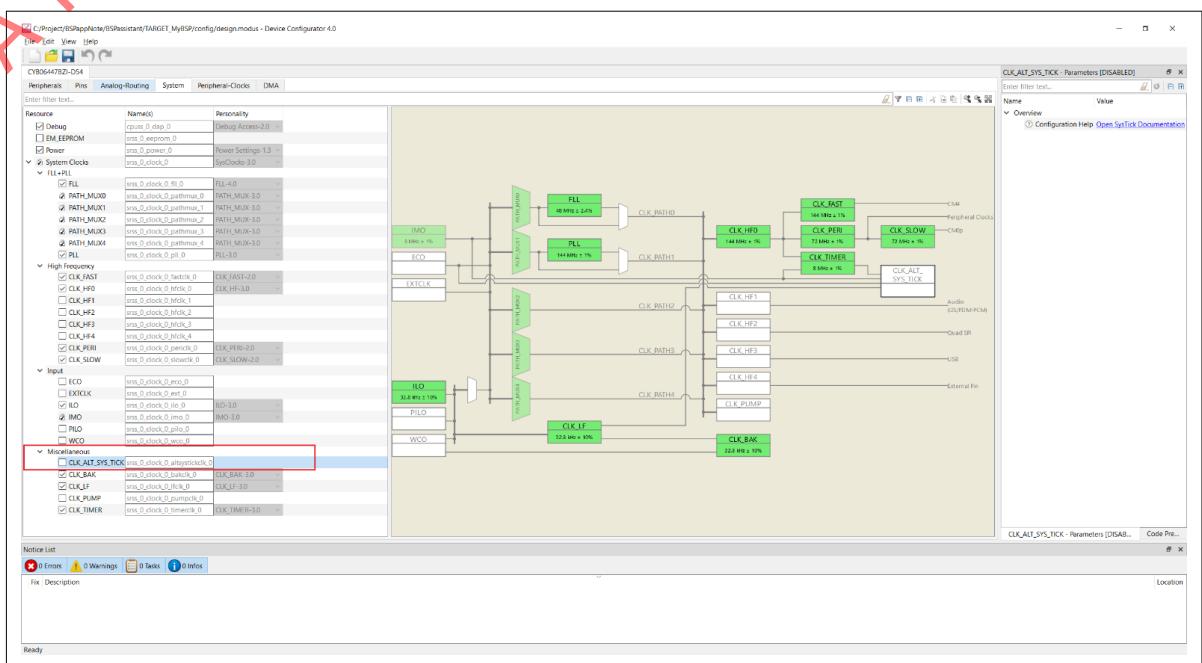


Figure 629 Changing clock settings

10. Click **File > Update All Personalities** and then click **File > Save**.
11. Close the Device Configurator tool.
12. On the BSP Assistant tool, under the **Dependencies** section, check that the BSP includes the following dependent libraries that will be used for an application:

- cat1cm0p
- core-lib
- core-make
- mtb-hal-cat1
- mtb-pdl-cat1
- receipte-make-cat1a

Note: If you click the filter button, only enabled dependencies will be listed.

5 PSoC™ 6 application notes

DRAFT

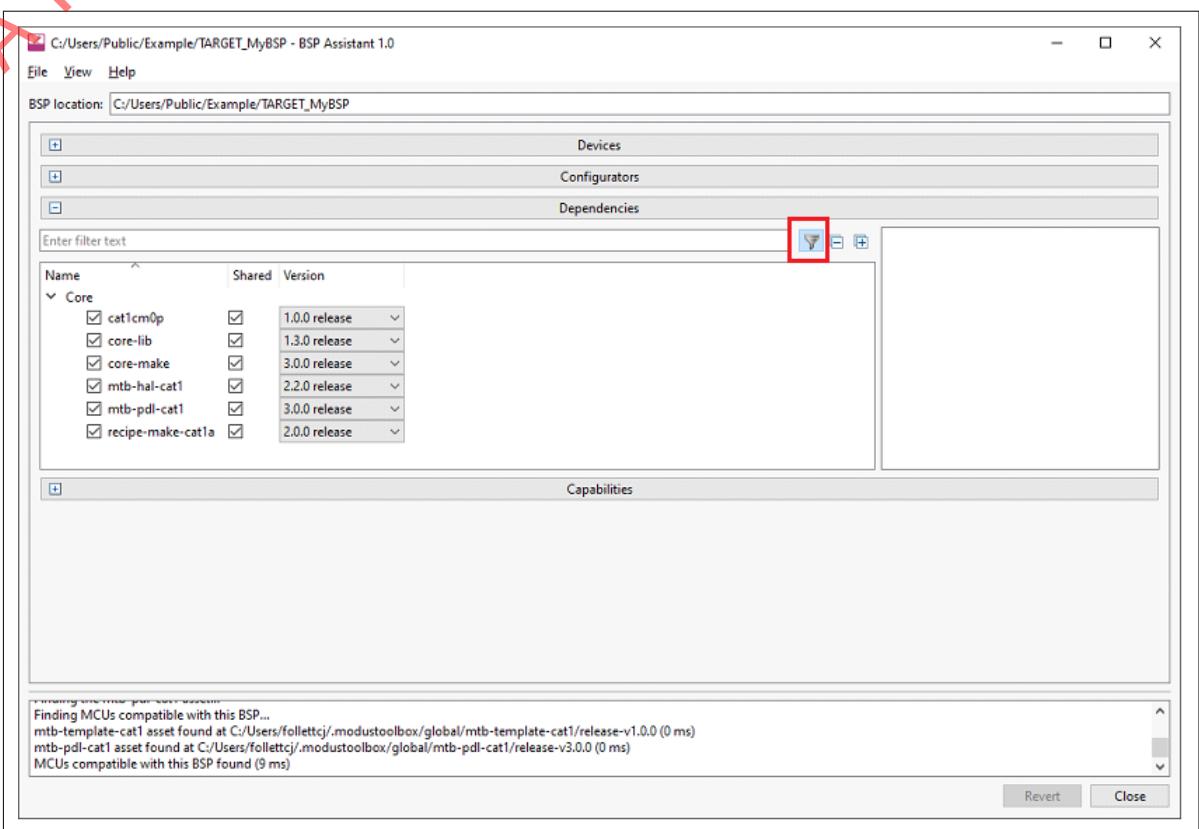


Figure 630 Checking dependencies

- 13.** If you need any additional **Dependencies** other than those listed above, disable the **Filter** icon in the **Dependencies** section to show other available dependencies. Enable the checkbox next to the dependency you wish to add. For example, the following image shows adding the **emwin** package.

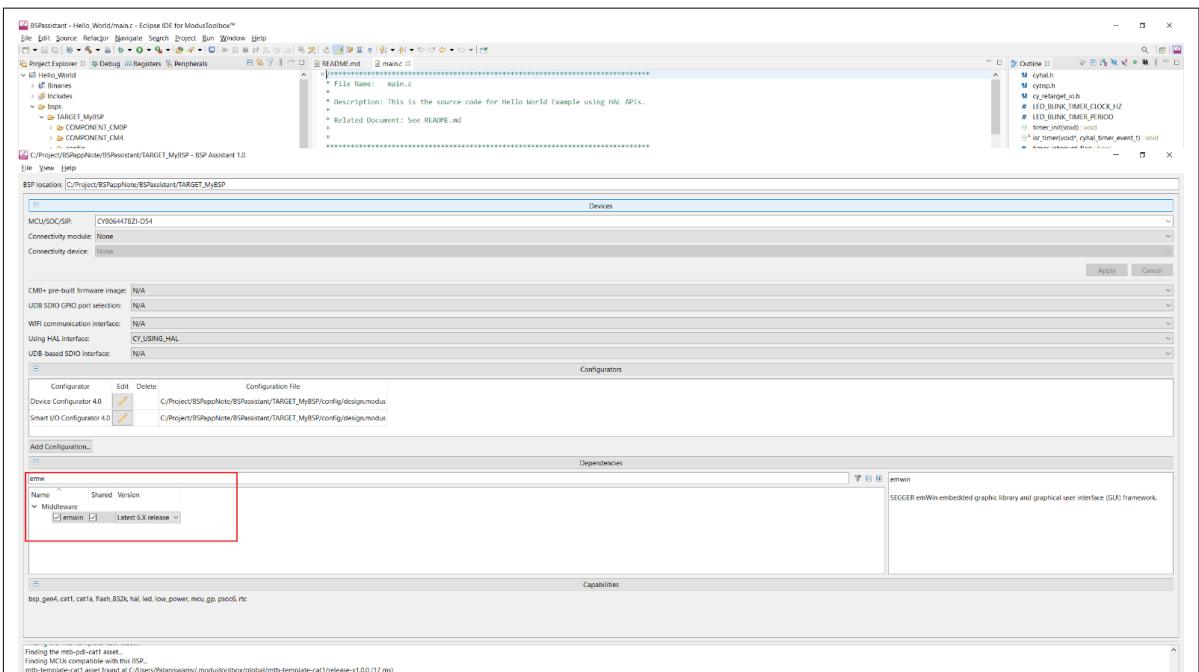


Figure 631 Adding "emwin" to the list of dependencies

- 14.** When creating an application based on a BSP, the project creator tool checks the capabilities required by the application against the capabilities provided by the BSP. The "Hello World" application provided on [github](#) expects the **hal** and **led** capabilities as per the application [manifest](#) file. The generic BSP that we

5 PSoC™ 6 application notes

DRAFT
used as a starting point has hal listed as a provided capability but not led because the generic BSP does not assume that an LED will be available on the board. This can be seen in the props.json file inside the BSP root folder:

```
"capabilities": [  
    "bsp_gen4",  
    "cat1",  
    "cat1a",  
    "hal",|  
    "low_power",  
    "mcu_gp",  
    "psoc6",  
    "rtc"
```

Figure 632 Missing led capability

15. Assume that the board for which the BSP is being created has an LED on it. You can update the props.json to add the led capability. You can also add the capability flash_832k as this device has on-chip flash memory of 812 KB:

```
"capabilities": [  
    "bsp_gen4",  
    "cat1",  
    "cat1a",  
    "flash_832k",  
    "hal",  
    "led",  
    "low_power",  
    "mcu_gp",  
    "psoc6",  
    "rtc"  
],
```

Figure 633 Added missing led and flash_832k capability

This completes the BSP creation; you can exit the tool now. If you changed any dependencies, you may see the following warning while exiting the tool; you can ignore the warning since the BSP is not yet being used in any applications:

5 PSoC™ 6 application notes

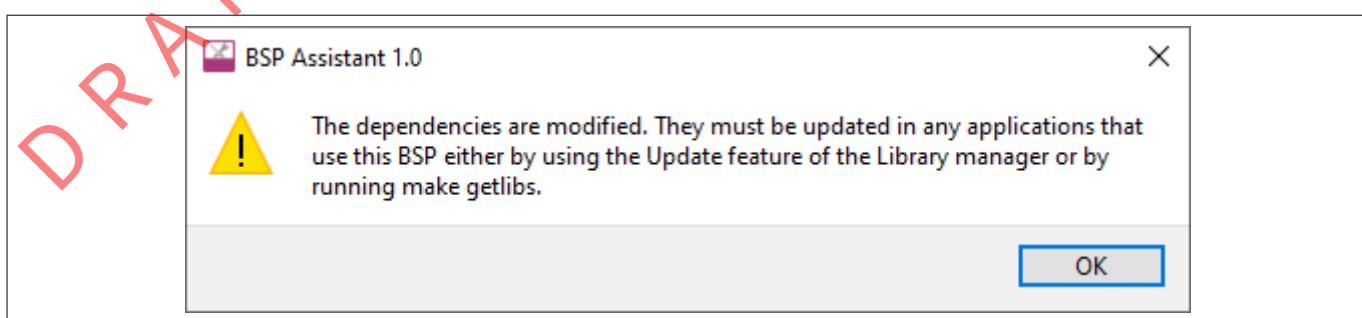


Figure 634 Warning can be ignored

5.19.3.1.2 Create an application

With the BSP creation and configuration complete, create a "Hello World" application from a code example template using the following steps.

1. Open Eclipse IDE for ModusToolbox™ from the Windows start menu and create a workspace as follows:

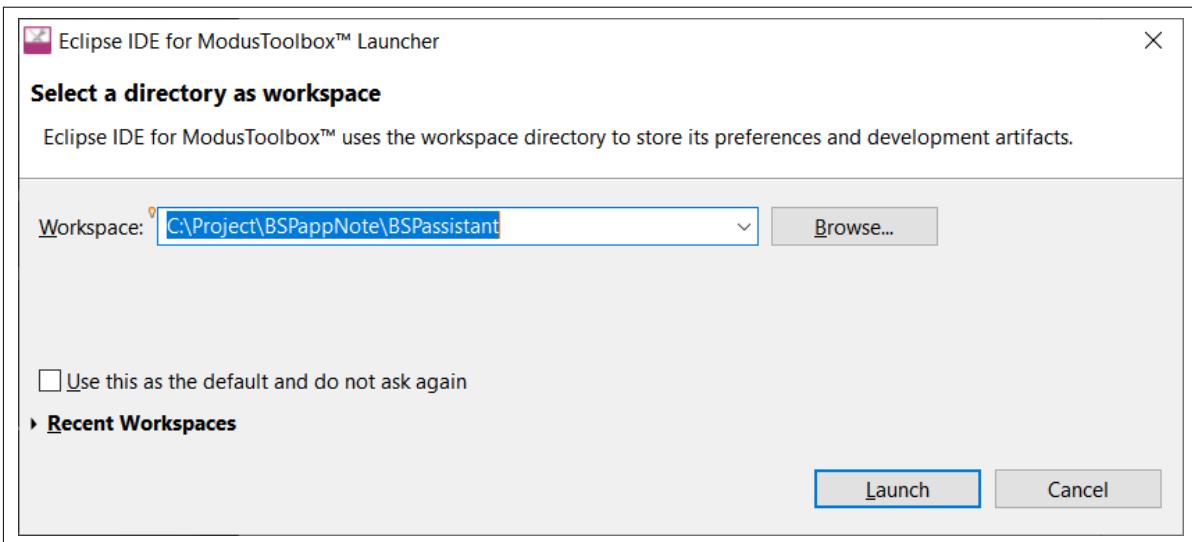


Figure 635 Create Eclipse IDE workspace

2. Click **File > New > ModusToolbox™ Application**.

5 PSoC™ 6 application notes

DRAFT

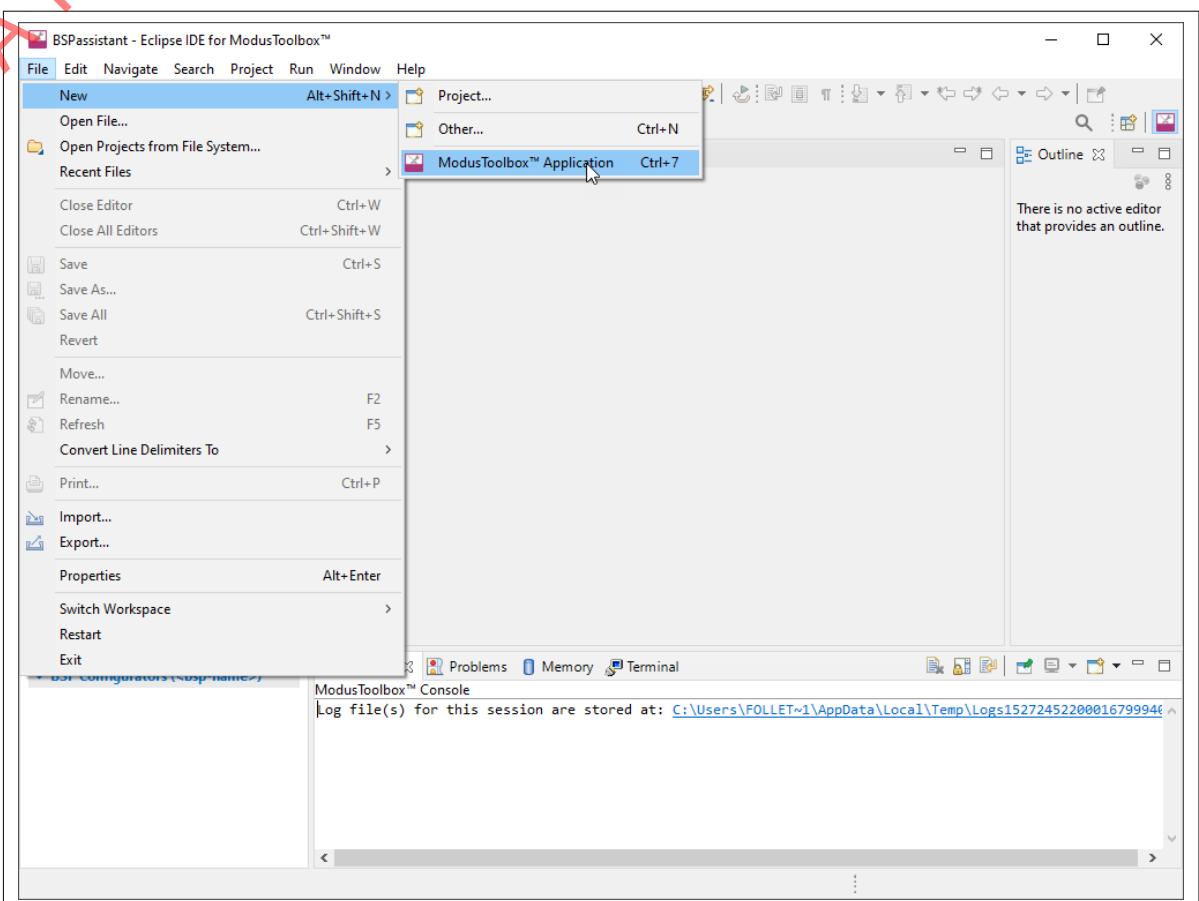


Figure 636 Create new ModusToolbox™ application

The Project Creator tool opens on the "Choose Board Support Package" dialog:

5 PSoC™ 6 application notes

DRAFT

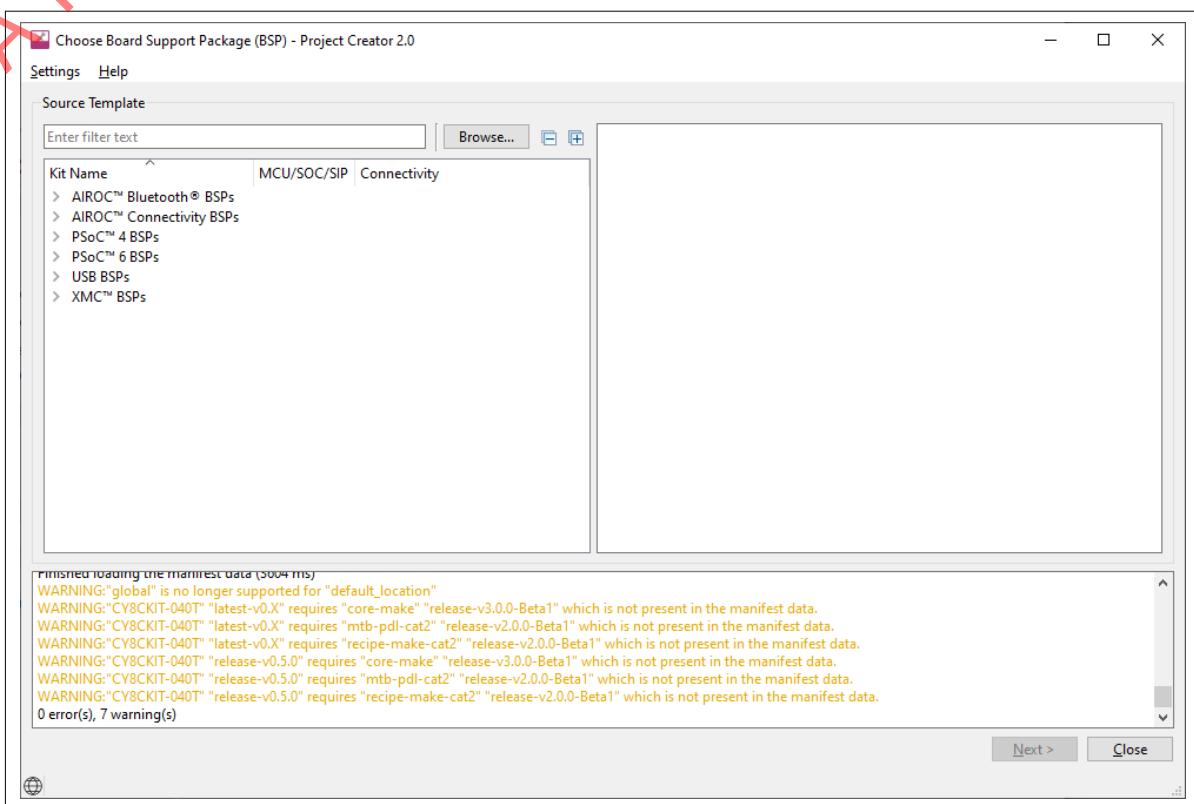


Figure 637 Project Creator tool

3. Select the BSP created in the previous section by clicking on the **Browse** button and navigating to the directory containing the previously created BSP and choosing it.

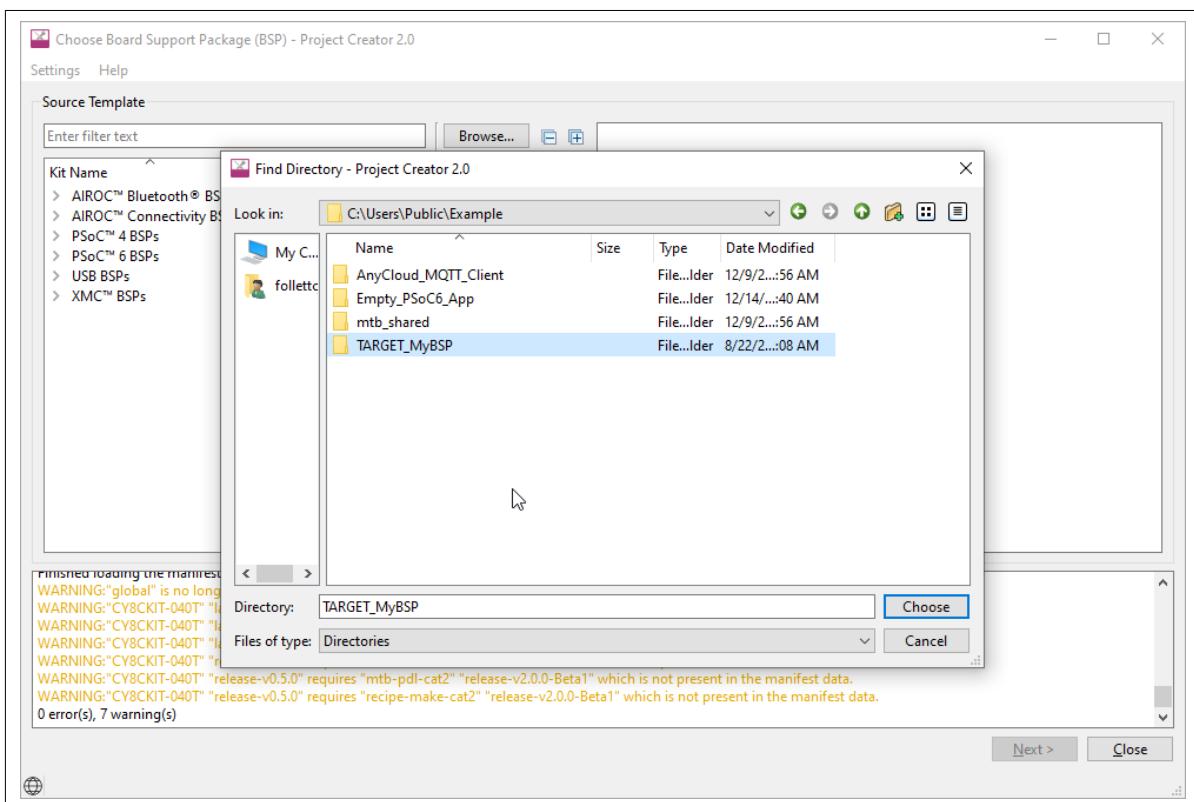


Figure 638 Importing BSP

4. The BSP you selected is listed as shown. Select it and click **Next >**.

5 PSoC™ 6 application notes

DRAFT

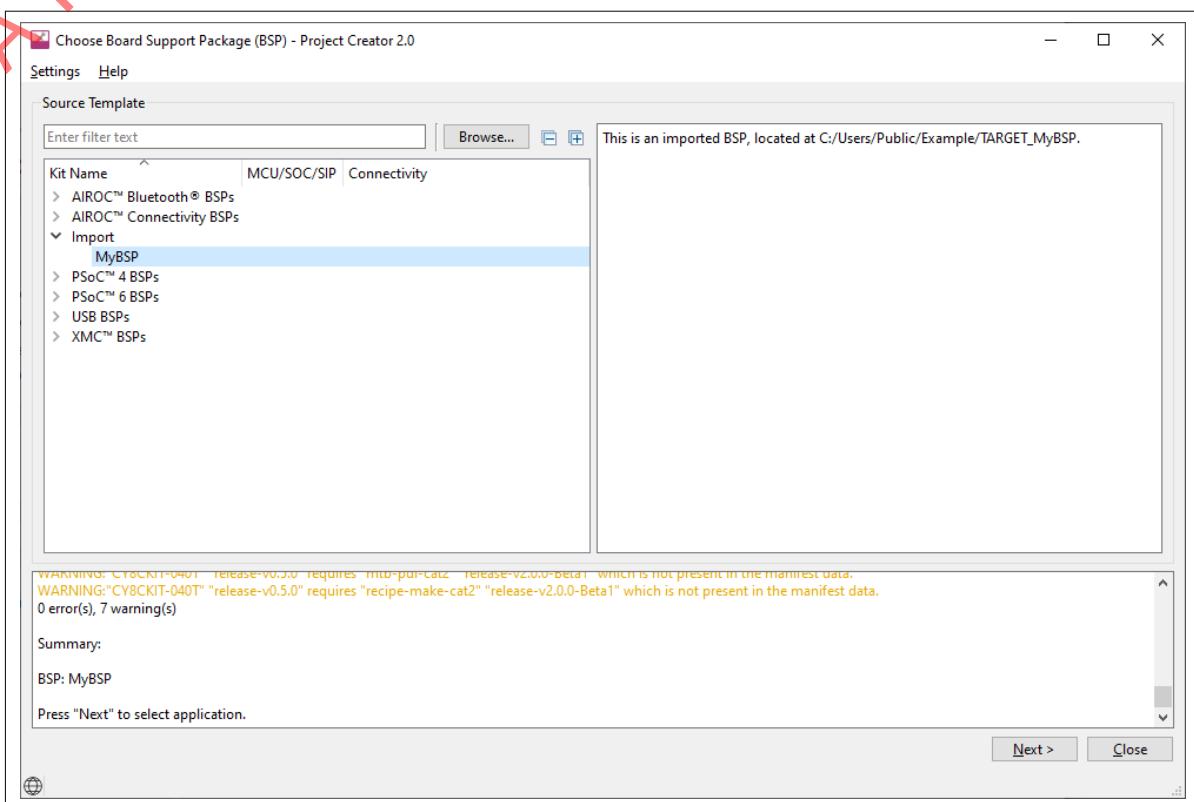


Figure 639 Selecting imported BSP

5. On the **Select Application** page, select **Hello World** from the template application under **Getting Started**. Enter a new name to the application such as **Hello_World** under the **New Application Name** column as shown. After that, click **Create**.

5 PSoC™ 6 application notes

DRAFT

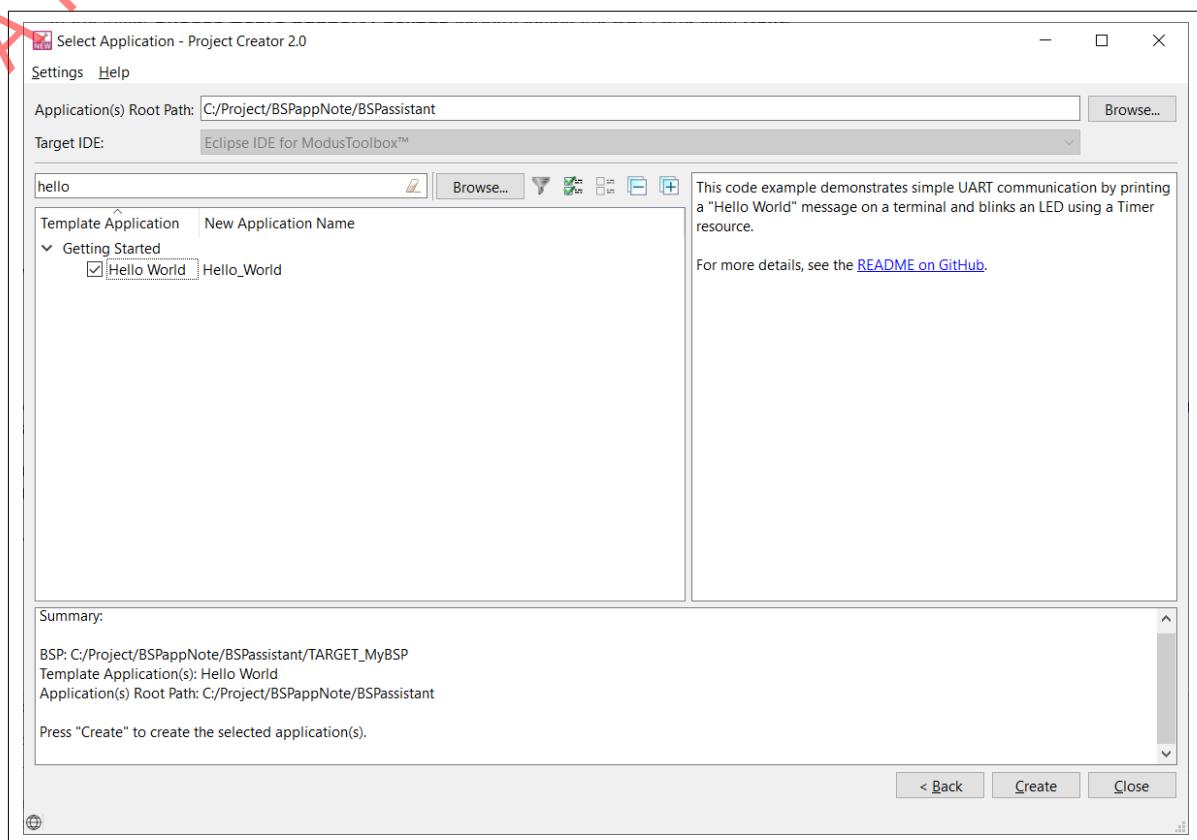


Figure 640 Selecting an application

The application download from GitHub starts and after successful completion, the project will be visible in the Eclipse IDE Project Explorer as follows:

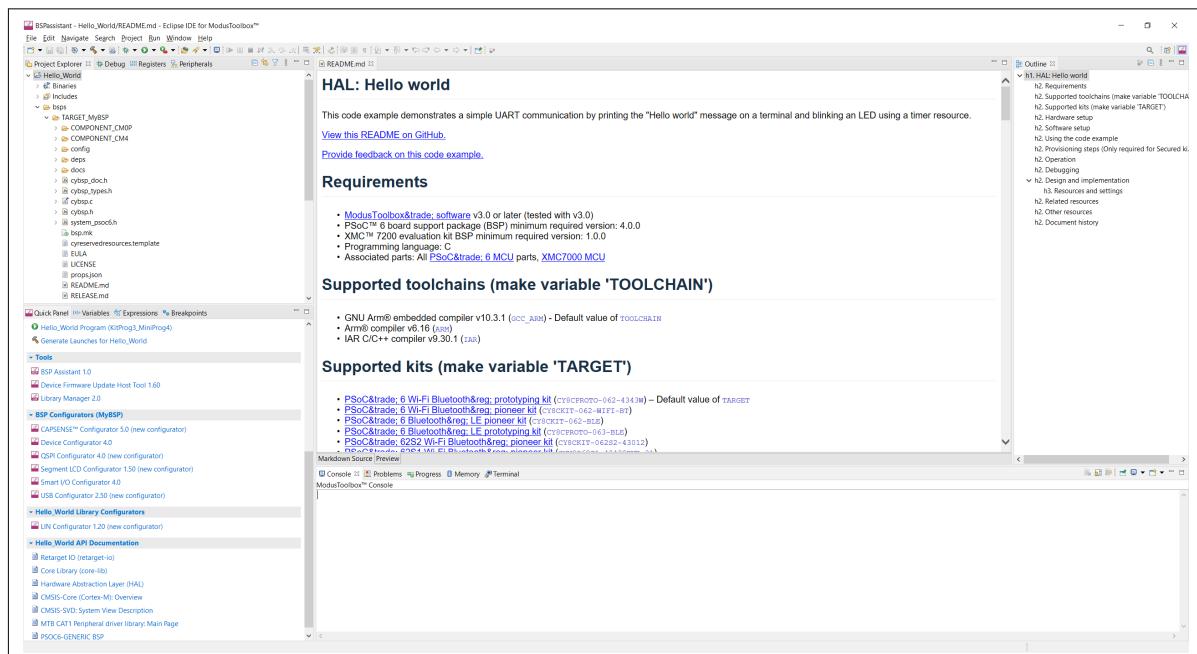


Figure 641 New application in Eclipse IDE for ModusToolbox™

5 PSoC™ 6 application notes

5.19.3.1.3 Code Build

1. Click **Build Application** in the **Quick Panel** pane as follows. If there are no errors, the build should pass successfully.

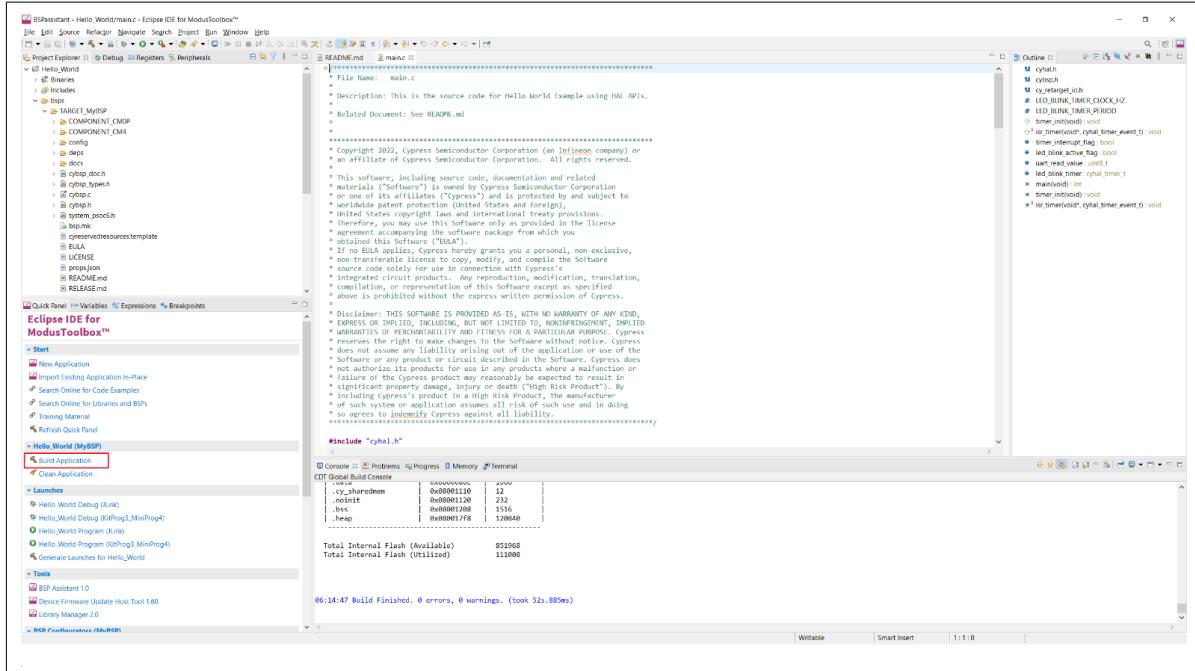


Figure 642 **Successful build**

This completes the use case demonstration.

5.19.3.2 Customizing an existing BSP

This use case happens when you want to make changes to an existing BSP that is included with the ModusToolbox™ ecosystem or to a BSP that was developed in a previous user project. For demonstration, this section customizes a BSP for the CY8CPROTO-062S3-4343W board by changing the MCU device to CY8C6347BZI-BLD43 instead of the built-in device (CY8C6245LQI-S3D72). The following steps show how to customize the BSP using the "Hello World" example from GitHub.

5.19.3.2.1 Create an application

1. Open Eclipse IDE for ModusToolbox™ from the Windows start menu and create a workspace as follows:

5 PSoC™ 6 application notes

DRAFT

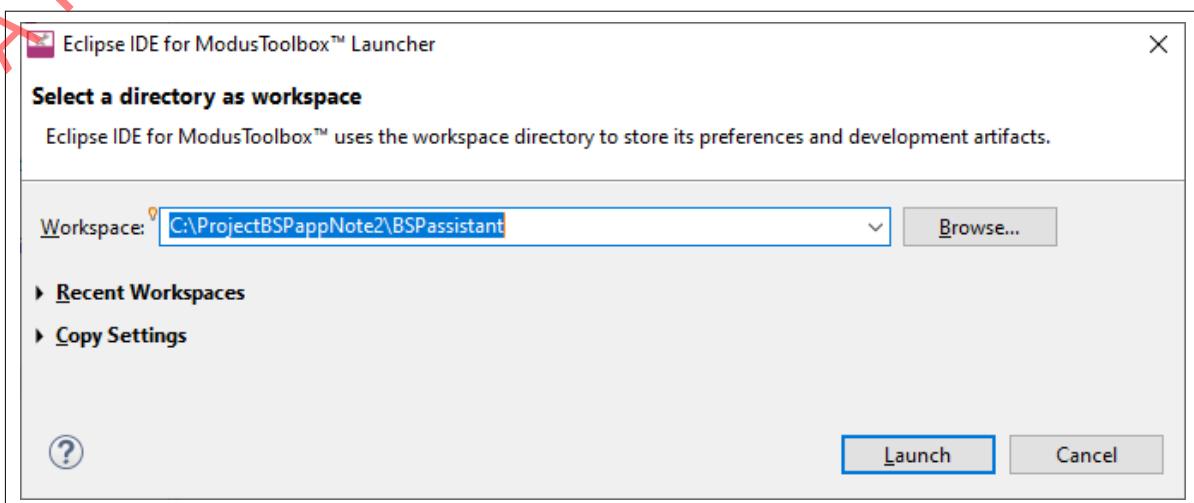


Figure 643 Create Eclipse IDE workspace

2. Click File > New > ModusToolbox™ Application.

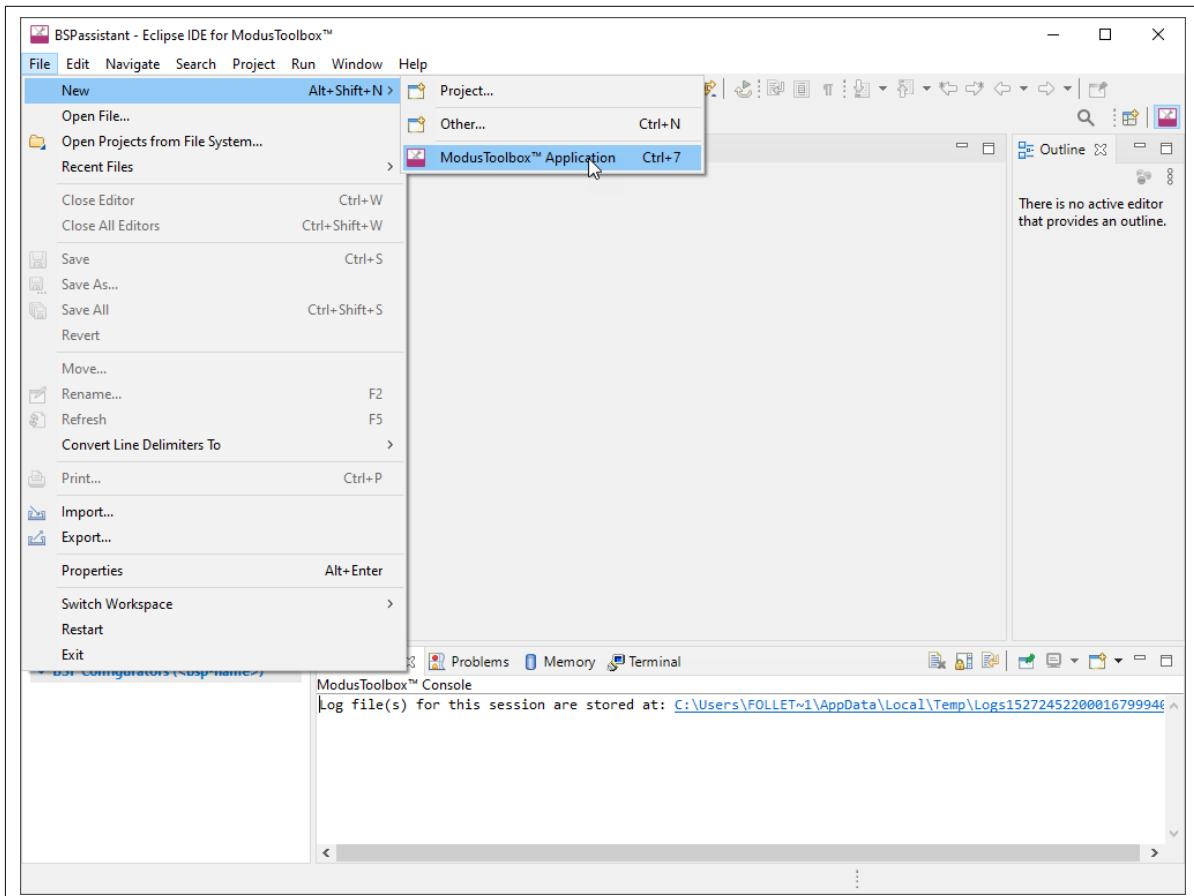


Figure 644 Create new application

3. The Project Creator tool opens; select the BSP as **CY8CPROTO-062S3-4343W** and click **Next >**.

5 PSoC™ 6 application notes

DRAFT

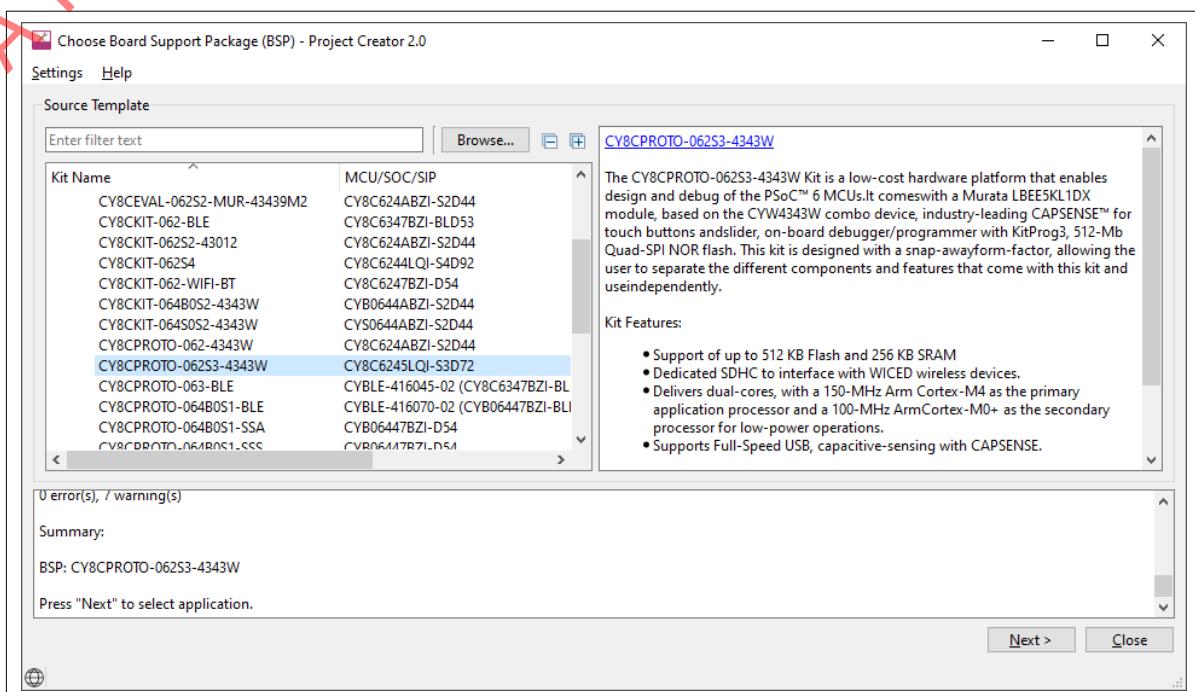


Figure 645 Selecting BSP

- On the **Select Application** page, select the **Hello World** template application under the **Getting Started** section as follows, and then click **Create**.

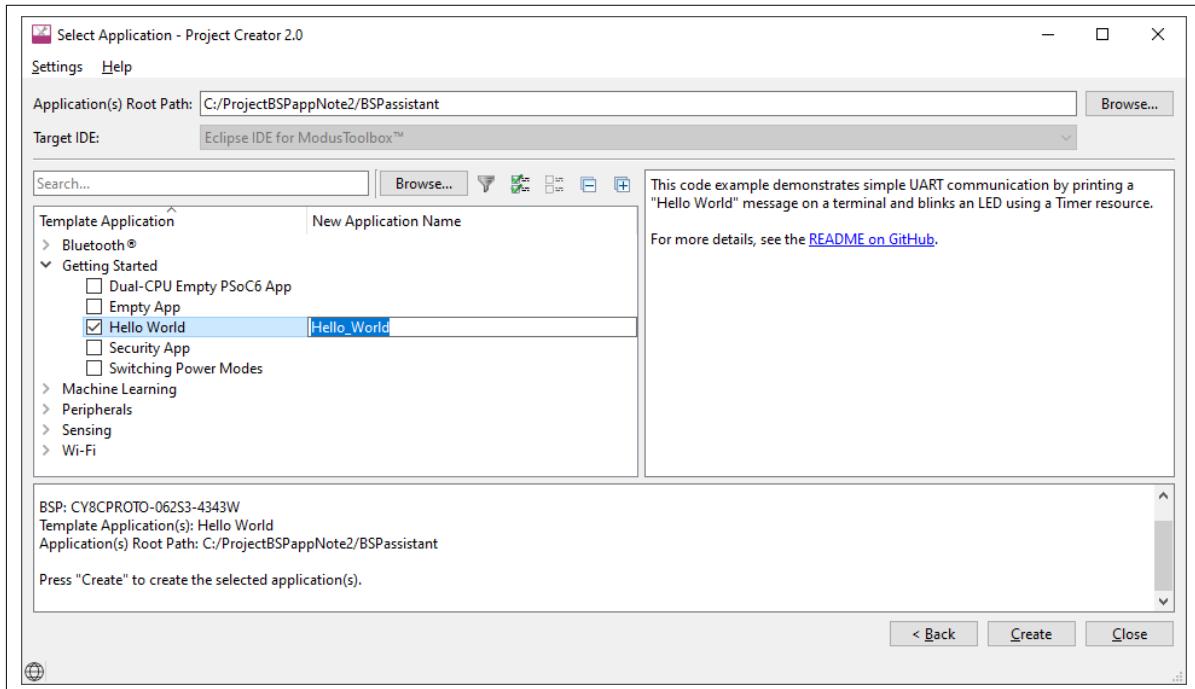


Figure 646 Selecting application

5.19.3.2.2 Customize the BSP

- When the application finishes loading in the Eclipse IDE, you can run the **BSP Assistant** tool by clicking on the **BSP Assistant <version>** from **Quick Panel** as shown.

5 PSoC™ 6 application notes

DRAFT

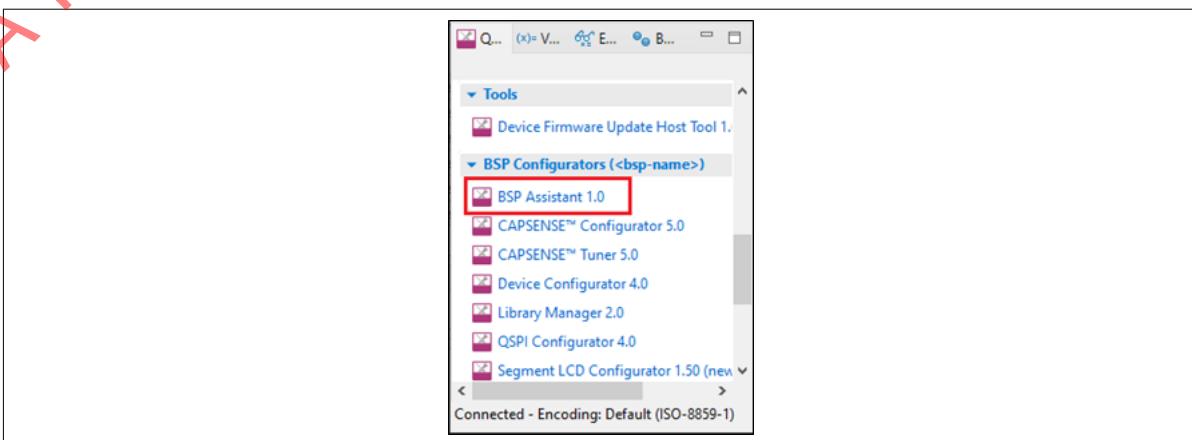


Figure 647 Launching BSP Assistant

- When the BSP Assistant loads, note that the MCU device being listed is **CY8C6245LQI-S3D72** which you can change to the new MCU device (**CY8C6347BZI-BLD43**) by using the pull-down menu.

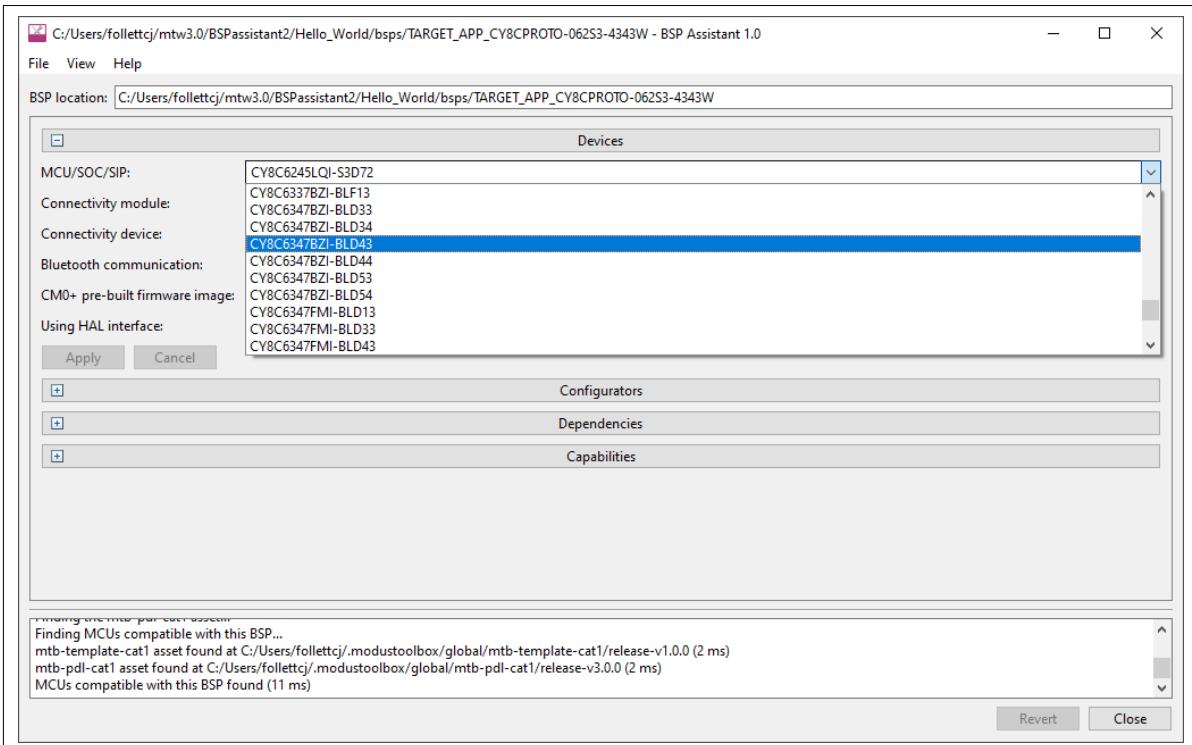


Figure 648 Changing MCU device

- Click **Apply** after the change. If the device change is successful, the "Device changed" message will be shown in the log as follows:

5 PSoC™ 6 application notes

DRAFT

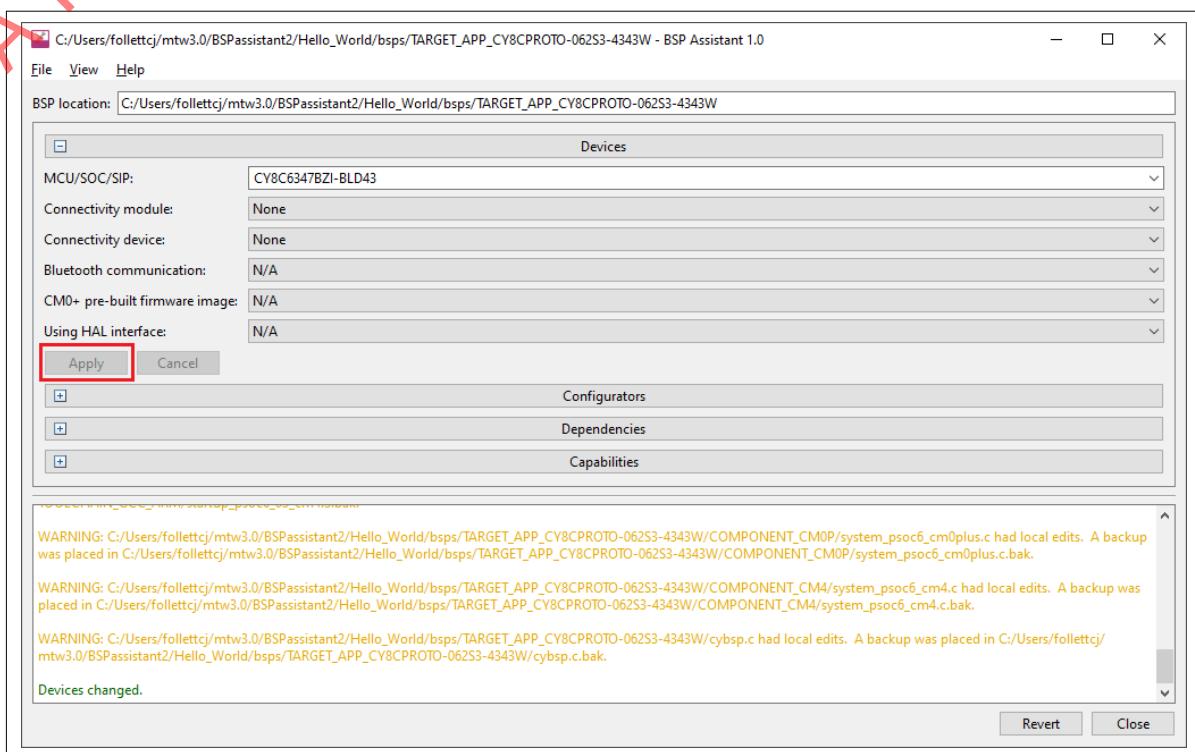


Figure 649 Applying change

4. It is a good practice to open **Device Configurator** to verify that the settings are appropriate for the new MCU.
5. This completes the BSP customization; you can now close the BSP Assistant tool.

5.19.3.2.3 Update dependencies and build

1. Run **Library Manager** by clicking **Library Manager <version>** from the **Quick Panel** in Eclipse IDE for ModusToolbox™ to fetch any new dependencies because the MCU was changed.

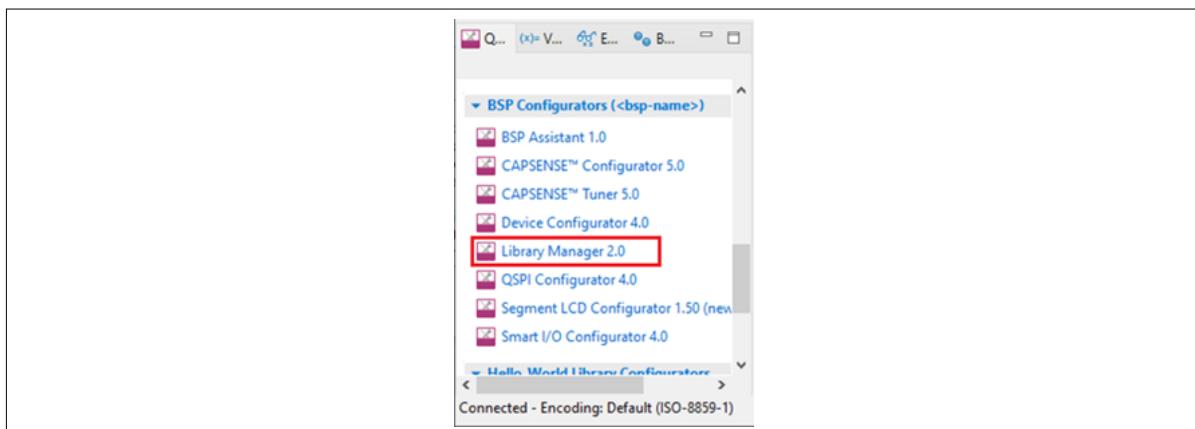


Figure 650 Launching Library Manager

2. In the **Library Manager** GUI, click **Update** to fetch the dependencies. This will start downloading the changes necessary for the BSP update.

5 PSoC™ 6 application notes

DRAFT

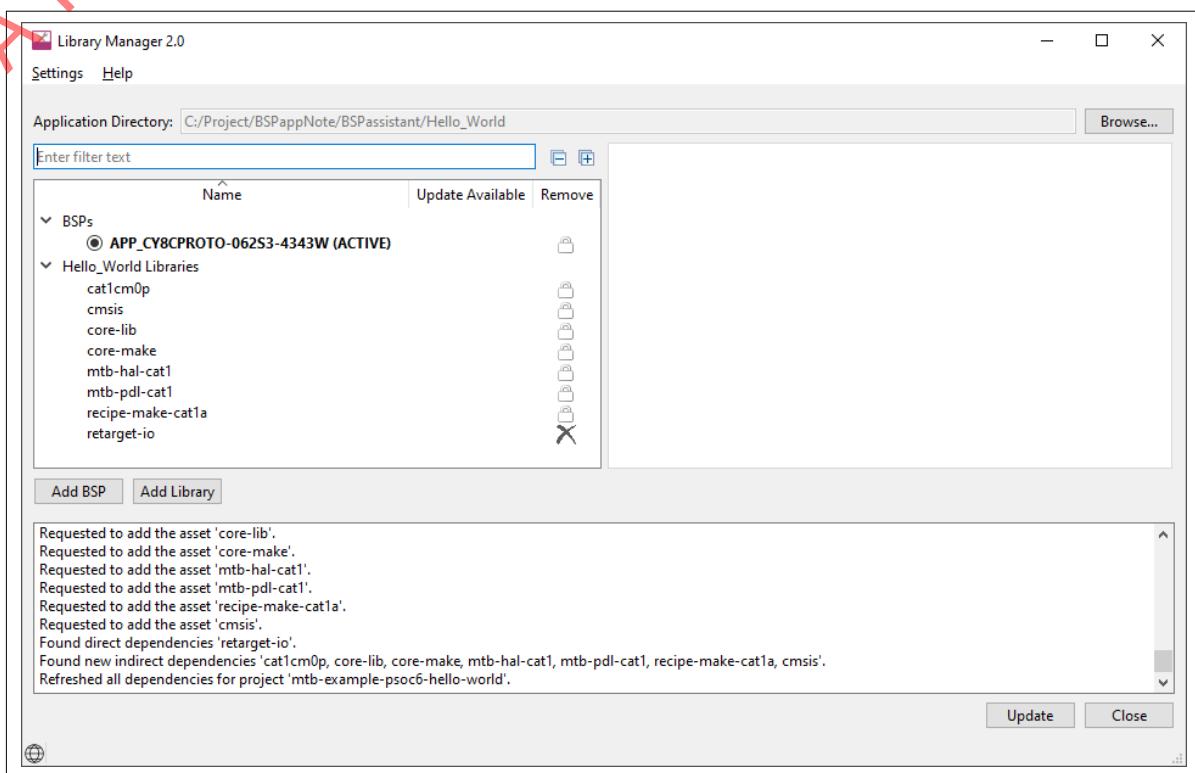


Figure 651 Updating the application

3. After the update is over, close the **Library Manager**.
4. In **Eclipse IDE for ModusToolbox™**, click **Build application** from the **Quick Panel**. If there are no errors, the build should pass successfully.

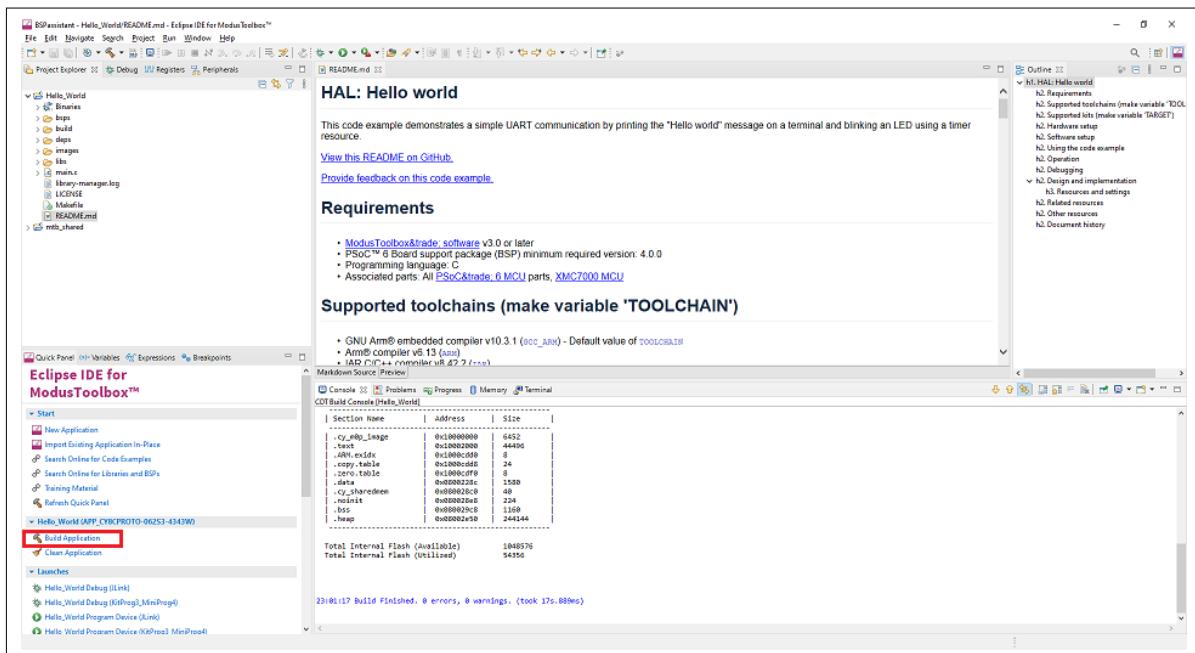


Figure 652 Successful build

This completes the use case demonstration.

~~5 PSoC™ 6 application notes~~

~~DRAFT~~ 5.19.4 Advanced usage

ModusToolbox™ BSPs that are either custom to a user application or preconfigured made available in [GitHub](#) can belong to different generations. You can find out the generation the BSP through the following:

Table 152 **BSP generations**

BSP root directory contains the file	BSP Generation
version.xml and not props.json	3
props.json and not version.xml	4

The BSP Assistant is only available for generation 4 BSPs using ModusToolbox™ version 3.0 or later.

Please note that BSP generation and BSP version have different meaning and should not be used interchangeably.

5.19.4.1 Differences between ModusToolbox™ BSP generations

BSP generation tools version 3 and 4 differ in the following ways:

Table 153 **BSP generation 3 and 4 differences**

Property	BSP generation 3	BSP generation 4
BSP configuration directory name	COMPONENT_BSP_DESIGN_MODUS	Config
BSP build toolchain linker file name	<platform>_<cpu>.sct for Arm® <platform>_<cpu>.ld for GCC <platform>_<cpu>.icf for IAR	linker.sct for Arm® linker.ld for GCC linker.icf for IAR
BSP version information	version tag in version.xml file in the BSP root directory	version field in props.json file in the BSP root directory
BSP makefile name	<bsp_name>.mk	bsp.mk
BSP locate_recipe.mk file	Present	Not present

Further, the contents of the Makefile differs significantly between the generations and the below figure shows a sample comparison:

5 PSoC™ 6 application notes

Figure 653 BSP Makefile Differences

5.19.4.2 Migrating the ModusToolbox™ BSP

ModusToolbox™ custom BSPs generated through version 2.x tools can be migrated to version 3.x tools using one of the following two methods so that the user application can take advantage of the latest features of the ModusToolbox™ ecosystem.

5.19.4.2.1 Using the BSP Assistant tool

1. Create a new BSP with the BSP Assistant tool by following the steps described in the [Creating a new BSP](#) section of this document.
 2. Update the following files in the newly created custom BSP with the settings from the corresponding version 2.x files:
 - BSP configuration files including `design.modus`, etc.
 - BSP linker files
 - Add any other dependencies for the BSP other than the default dependencies by using the **Dependencies** section in the BSP Assistant tool
 3. Import the newly created custom BSP into your ModusToolbox™ application by running the Library Manager tool in your ModusToolbox™ application directory by clicking **Add BSP** and then clicking **Browse** on the Add or Import BSP dialog as follows:

5 PSoC™ 6 application notes

DRAFT

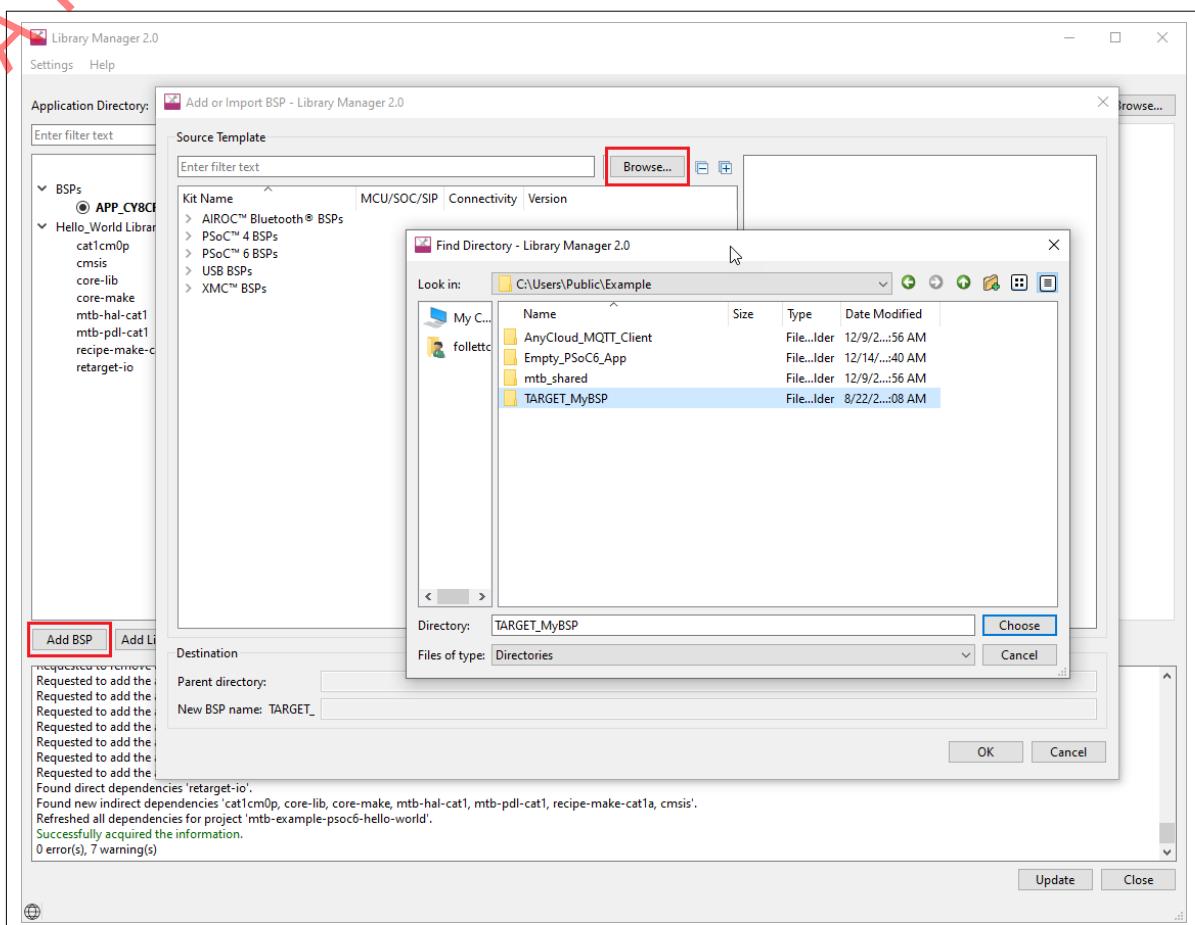


Figure 654

4. Make the new BSP active.

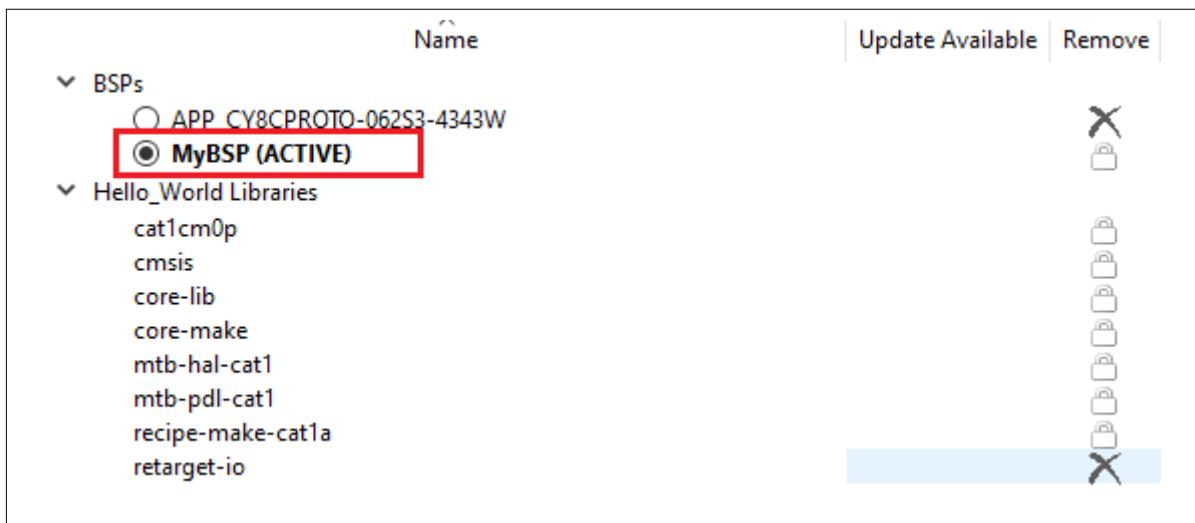


Figure 655

This completes the migration of the BSP from version 2.x to 3.x using the BSP Assistant tool.

5.19.4.2.2 Without using the BSP Assistant tool

1. Delete the version.xml file.
2. Delete the deps directory.

5 PSoC™ 6 application notes

- ~~DRAFT~~
3. Create a new file called `props.json` and add the various properties by looking into a similar file from preconfigured 3.x BSPs in [GitHub](#).
 4. Rename the BSP configuration directory `COMPONENT_BSP_DESIGN_MODUS` to `config`.
 5. Rename the BSP makefile `<bsp_name>.mk` to `bsp.mk`.
 6. Update the `bsp.mk` file by comparing with an existing 3.x format `bsp.mk` file because there are significant changes between the 2 generations.
 7. Rename the linker files in the directory `COMPONENT_CM0P` and `COMPONENT_CM4` as follows:
 - `<platform>_<cpu>.sct` to `linker.sct` for Arm®
 - `<platform>_<cpu>.ld` to `linker.ld` for GCC
 - `<platform>_<cpu>.icf` to `linker.icf` for IAR
 8. Delete the `locate_recipe.mk` file.
 9. Follow the steps in [Section 3.1.2](#) to import the new BSP to the application.

This completes the migration of the BSP from version 2.x to 3.x without using the BSP Assistant tool.

5 PSoC™ 6 application notes

References

-
- 1
- 9
- 1. [ModusToolbox™ home page](#)
- 2. [AN228571 - Getting started with PSoC™ 6 MCU on ModusToolbox™ software](#)
- 3. [ModusToolbox™ tools package user guide](#)
- 4. [ModusToolbox™ BSP Assistant user guide](#)
- 5. [Creating Custom BSPs in ModusToolbox - KBA230822 \(Specific to version 2.x BSPs\)](#)
- 6. [Migrating ModusToolbox™ applications from version 2.x to version 3.x - KBA236134](#)

~~6 PSoC™ 6 development kits~~

~~5.19.5 Revision history~~

Document version	Date of release	Description of changes
**	2022-10-18	Initial release.

6 PSoC™ 6 development kits

Infineon provides a broad portfolio of hardware development kits to aid in the rapid prototyping and evaluation of PSoC™ 6 MCU based applications. All the kits are supported in the ModusToolbox™ software through board support packages (BSP) and hundreds of validated code examples.

The list of kits and the relevant links are listed on this page.

Note: *To evaluate an application targeting the PSOC™ 61 MCU Programmable line, choose the corresponding PSOC™ 62 MCU development kit, and use only the ARM® Cortex®-M4+ CPU of the dual core PSOC™ 62 MCU for running the application code. The PSOC™ 62 MCU will then emulate the functionality of the single core ARM® Cortex®-M4 PSOC™ 61 MCU.*

Kit MPN	Applicable PSoC™ 6 families	Kit Type	Documents	Hardware Design Files	BSP GitHub Repository
---------	-----------------------------	----------	-----------	-----------------------	-----------------------

Kits for PSoC™ 61 Programmable line, PSoC™ 62 Performance line

CY8CKIT-062S4	CY8C61x4, CY8C62x4	Pioneer	Quick Start Guide , User Guide	Zip file	BSP
CY8CPROTO-062S3-4343W	CY8C61x5, CY8C62x5	Proto	Quick Start Guide , User Guide	Zip file	BSP
CY8CKIT-062-WIFI-BT	CY8C61x6, CY8C61x7, CY8C62x6, CY8C62x7	Pioneer	Quick Start Guide , User Guide	Zip file	BSP
CY8CPROTO-062-4343W	CY8C61x8, CY8C61xA, CY8C62x8, CY8C62xA	Proto	Quick Start Guide , User Guide	Zip file	BSP
CY8CKIT-062S2-43012	CY8C61x8, CY8C61xA, CY8C62x8, CY8C62xA	Pioneer	Quick Start Guide , User Guide	Zip file	BSP
CY8CEVAL-062S2	CY8C61x8, CY8C61xA, CY8C62x8, CY8C62xA	Evaluation	Quick Start Guide , User Guide	Zip file	BSP

Kits for PSoC™ 63 Bluetooth® Low Energy Connectivity line

CY8CKIT-062-BLE	CY8C63x6, CY8C63x7	Pioneer	Quick Start Guide , User Guide	Zip file	BSP
-----------------	-----------------------	---------	--	--------------------------	---------------------

~~7 Other PSoC™ 6 documents~~

Kit MPN	Applicable PSoC™ 6 families	Kit Type	Documents	Hardware Design Files	BSP GitHub Repository
CY8CPROTO-063-BLE	CY8C63x6, CY8C63x7	Proto	Quick Start Guide, User Guide	Zip file	BSP

Kits for PSoC™ 64 Secure MCU line

CY8CPROTO-064-B0S3	CYB064x5	Proto	Quick Start Guide, User Guide	Zip file	BSP
CY8CPROTO-064S-1-SB	CYB064x7	Proto	Quick Start Guide, User Guide	Zip file	BSP
CY8CKIT-064B0S2-4343W	CYB064xA	Pioneer	Quick Start Guide, User Guide	Zip file	BSP
CY8CKIT-064S0S2-4343W	CYS064xA	Pioneer	Quick Start Guide, User Guide	Zip file	BSP

7 Other PSoC™ 6 documents

7.1 PSoC™ CAD, BSDL, and IBIS files

CAD, BSDL, and IBIS files can be used in various tools to help engineers design, test, and debug devices that integrate PSoC™ 6 products.

PSoC™ 6 CAD Library Files

PSoC™ 6 CAD libraries provide footprint and schematic support for common PCB design software like Allegro, Altium. CAD libraries for the various product families are listed in the table below.

PSoC™ 6 Product Family	CAD Library File
CY8C61x4, CY8C62x4	File link
CY8C61x5, CY8C62x5, CYB064x5	File link
CY8C61x6, CY8C61x7, CY8C62x6, CY8C62x7, CYB064x7	File link
CY8C61x8, CY8C61xA, CY8C62x8, CY8C62xA, CYB064xA, CYS064xA	File link

PSoC™ 6 BSDL Files

Boundary-Scan Description Language (BSDL) is a subset of VHDL (VHSIC Hardware Description Language) that describes how boundary-scan (JTAG) is implemented in a device and how the device operates. It defines the data transport characteristics of the device, how it captures, shifts, and updates scanned data. Boundary-scan tools usually require that the user supply BSDL files for the devices being used in order to properly generate test vectors and perform in-system programming and functional testing.

PSoC™ 6 BSDL files for the various product families are listed in the below table.

PSoC™ 6 Product Family	BSDL File
CY8C61x4, CY8C62x4	File link

~~7 Other PSoC™ 6 documents~~

PSoC™ 6 Product Family	BSDL File
CY8C61x5, CY8C62x5, CYB064x5	File link
CY8C61x6, CY8C61x7, CY8C62x6, CY8C62x7, CYB064x7	File link
CY8C61x8, CY8C61xA, CY8C62x8, CY8C62xA, CYB064xA, CYS064xA	File link

PSoC™ 6 IBIS Files

“Input/Output (I/O) Buffer Information Specification” (IBIS) is a text file that contains electrical characteristics of a device. The type of information provided in the file is useful for PC-board design where signal integrity and timing analysis is done to design a part into a system.

PSoC™ 6 IBIS files for the various product families are listed in the below table.

PSoC™ 6 Product Family	IBIS File
CY8C61x4, CY8C62x4	File link
CY8C61x5, CY8C62x5, CYB064x5	File link
CY8C61x6, CY8C61x7, CY8C62x6, CY8C62x7, CYB064x7	File link
CY8C61x8, CY8C61xA, CY8C62x8, CY8C62xA, CYB064xA, CYS064xA	File link

7.2 PSoC™ 6 technical reference manuals

A technical reference manual (TRM) provides detailed technical information on a device family. Architecture TRMs provide a functional description of the various sub-blocks in the device including block features, architecture, and use cases. Register TRMs provide a register level description of the user accessible registers in the device.

The TRMs for the PSoC™ 6 device families are listed below.

Device Family	Architecture TRM	Registers TRM
PSoC™ 61 Programmable Line (Single Core Applications CPU: ARM® Cortex®-M4)		
CY8C61x4	PDF	PDF
CY8C61x5	PDF	PDF
CY8C61x6, CY8C61x7	PDF	PDF
CY8C61x8, CY8C61xA	PDF	PDF
PSoC™ 62 Performance Line (Dual Core Applications CPU: ARM® Cortex®-M4, ARM® Cortex®-M0+)		
CY8C62x4	PDF	PDF
CY8C62x5	PDF	PDF
CY8C62x6, CY8C62x7	PDF	PDF
CY8C62x8, CY8C62xA	PDF	PDF
PSoC™ 63 Bluetooth® Low Energy Connectivity Line (MCUs with on-chip Bluetooth® radio)		
CY8C63x6, CY8C63x7	PDF	PDF
PSoC™ 64 Secure MCU Line (Applications CPU: ARM® Cortex®-M4, Secure CPU: ARM® Cortex®-M0+)		
CYB064x5 Secure Boot	PDF	PDF
CYB064x7 Secure Boot	PDF	PDF

7 Other PSoC™ 6 documents

Device Family	Architecture TRM	Registers TRM
CYB064xA Secure Boot	PDF	PDF
CYB064x7-BL Secure Boot BLE	PDF	PDF
CYS064xA Standard Secure	PDF	PDF

Related information

[PSoC 6 datasheets](#) on page 29

Trademarks

All referenced product or service names and trademarks are the property of their respective owners.

Edition 2023-03-08

Published by

**Infineon Technologies AG
81726 Munich, Germany**

**© 2023 Infineon Technologies AG
All Rights Reserved.**

Do you have a question about any aspect of this document?

Email: erratum@infineon.com

**Document reference
IFX-lhy1649327762448**

Important notice

The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics ("Beschaffenheitsgarantie").

With respect to any examples, hints or any typical values stated herein and/or any information regarding the application of the product, Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind, including without limitation warranties of non-infringement of intellectual property rights of any third party.

In addition, any information given in this document is subject to customer's compliance with its obligations stated in this document and any applicable legal requirements, norms and standards concerning customer's products and any use of the product of Infineon Technologies in customer's applications.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer's technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

Warnings

Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies' products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.