Debugging with PSoC® 1

Author: Dan Sweet

Associated Part Family: All PSoC1 Families
Related Application Notes: None

To get the latest version of this application note, or the associated project file, please visit http://www.cypress.com/go/AN73212.

AN73212 introduces the elements of the PSoC® 1 debugger system and explains how to configure and use them effectively. Several common debugging techniques are described to help you solve common problems, such as stack overflow and memory corruption. A troubleshooting guide is included.

Contents

1 Introduction

The purpose of this application note is to introduce the hardware and software debugger elements available in PSoC1 and to describe several common debugging techniques.

The primary hardware elements of the debugging system are an In-Circuit Emulator (ICE) and a debug pod with an on chip debugger (OCD) enabled PSoC1 device. Those elements, and instructions on configuring and using them, are described in the Debugging Hardware section of this application note.

The software elements center on PSoC Designer®.

This integrated development environment has many tools for debugging including breakpoints, watch variables, memory viewer, trace, events, and output files (list and map). Each of these elements is described in the Debugging Environment – PSoC Designer IDE section of this application note.

In addition, the following topics are discussed:

- PSoC1 Debugging Tips and Tricks
- Alternative Debugging Options
- Troubleshooting
- PSoC1 Debugging Tips and Tricks
- Alternative Debugging Options
- Troubleshooting

2 PSoC Resources

Cypress provides a wealth of data at www.cypress.com to help you to select the right PSoC device for your design, and quickly and effectively integrate the device into your design. In this document, PSoC refers to the PSoC 1 family of devices. To learn more about PSoC 1, refer to the application note AN75320 - *Getting Started with PSoC 1*.

The following is an abbreviated list for PSoC 1:

- **Overview:** PSoC Portfolio, PSoC Roadmap
- **Product Selectors:** PSoC 1, PSoC 3, PSoC 4, or PSoC 5LP. In addition, PSoC Designer includes a device selection tool.
- **Datasheets:** Describe and provide electrical specifications for the PSoC 1 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.
- **Technical Reference Manuals (TRM):** Provide detailed descriptions of the internal architecture of the PSoC 1 devices.
- **Development Kits:**
  - CY3215A-DK In-Circuit Emulation Lite Development Kit includes an in-circuit emulator (ICE). While the ICE-Cube is primarily used to debug PSoC 1 devices, it can also program PSoC 1 devices using ISSP.
  - CY3210-PSOCEVAL1 Kit enables you to evaluate and experiment Cypress's PSoC 1 programmable system-on-chip design methodology and architecture.
  - CY8CKIT-001 is a common development platform for all PSoC family devices.
- The MiniProg1 and MiniProg3 devices provide an interface for flash programming.

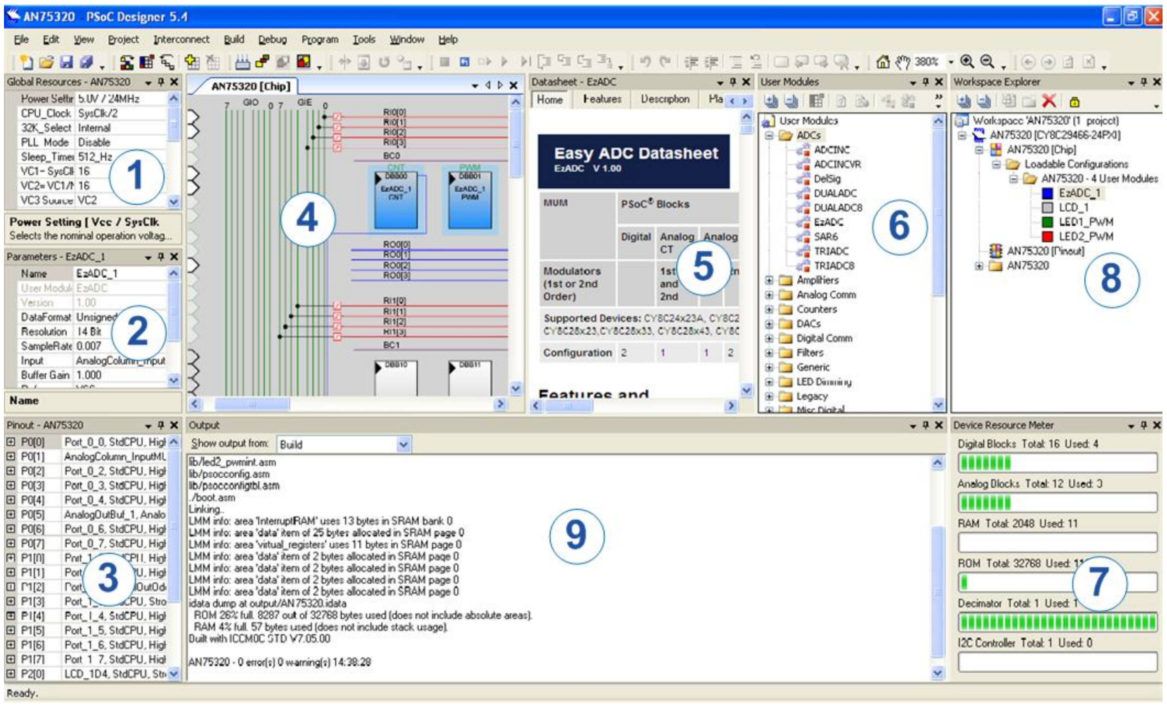
2.1 PSoC Designer

PSoC Designer is a free Windows-based Integrated Design Environment (IDE). Develop your applications using a library of pre-characterized analog and digital peripherals in a drag-and-drop design environment. Then, customize your design leveraging the dynamically generated API libraries of code. Figure 1 shows PSoC Designer windows. **Note:** This is not the default view.

1. **Global Resources –** all device hardware settings.
2. **Parameters –** the parameters of the currently selected User Modules.
3. **Pinout –** information related to device pins.
4. **Chip-Level Editor –** a diagram of the resources available on the selected chip.
5. **Datasheet –** the datasheet for the currently selected UM
6. **User Modules –** all available User Modules for the selected device.
7. **Device Resource Meter –** device resource usage for the current project configuration.
8. **Workspace –** a tree level diagram of files associated with the project.
9. **Output –** output from project build and debug operations.

Note: For detailed information on PSoC Designer, go to **PSoC® Designer > Help > Documentation > Designer Specific Documents > IDE User Guide**.

Figure 1. PSoC Designer Layout



Power Settir	5.0V / 24MHz
CPU_Clock	SysClk/2
32K_Select	Internal
PLL Mode	Disable
Sleep_Timer	512_Hz
VC1= SysCl	16
VC2= VC1/N	16
VC3 Source	VC2

Power Setting [Vcc / SysClk
Selects the nominal operation voltag...

Name	EzADC_1
User Modul	EzADC
Version	1.00
DataFormat	Unsigned
Resolution	14 Bit
SampleRate	0.007
Input	AnalogColumn_Input
Buffer Gain	1.000

Show output from: Build

```
lib/led2_pwmint.asm
lib/psocconfig.asm
lib/psocconfigtbl.asm
./boot.asm
Linking..
LMM info: area 'InterruptRAM' uses 13 bytes in SRAM bank 0
LMM info: area 'data' item of 25 bytes allocated in SRAM page 0
LMM info: area 'virtual_registers' uses 11 bytes in SRAM page 0
LMM info: area 'data' item of 2 bytes allocated in SRAM page 0
LMM info: area 'data' item of 2 bytes allocated in SRAM page 0
LMM info: area 'data' item of 2 bytes allocated in SRAM page 0
LMM info: area 'data' item of 2 bytes allocated in SRAM page 0
idata dump at output/AN75320.idata
  ROM 26% full. 8287 out of 32768 bytes used (does not include absolute areas).
  RAM 4% full. 57 bytes used (does not include stack usage).
Built with ICCM8C STD V7.05.00

AN75320 - 0 error(s) 0 warning(s) 14:38:28
```

Digital Blocks Total: 16 Used: 4
Analog Blocks Total: 12 Used: 3
RAM Total: 2048 Used: 11
ROM Total: 32768 Used: 11
Decimator Total: 1 Used: 1
I2C Controller Total: 1 Used: 0

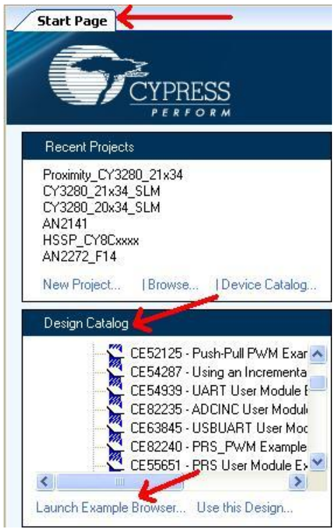
2.2 Code Examples

The following webpage lists the PSoC Designer based Code Examples. These Code Examples can speed up your design process by starting you off with a complete design, instead of a blank page and also show how PSoC Designer User modules can be used for various applications.

http://www.cypress.com/go/PSoC1Code Examples

To access the Code Examples integrated with PSoC Designer, follow the path **Start Page > Design Catalog > Launch Example Browser** as shown in Figure 2.

Figure 2. Code Examples in PSoC Designer



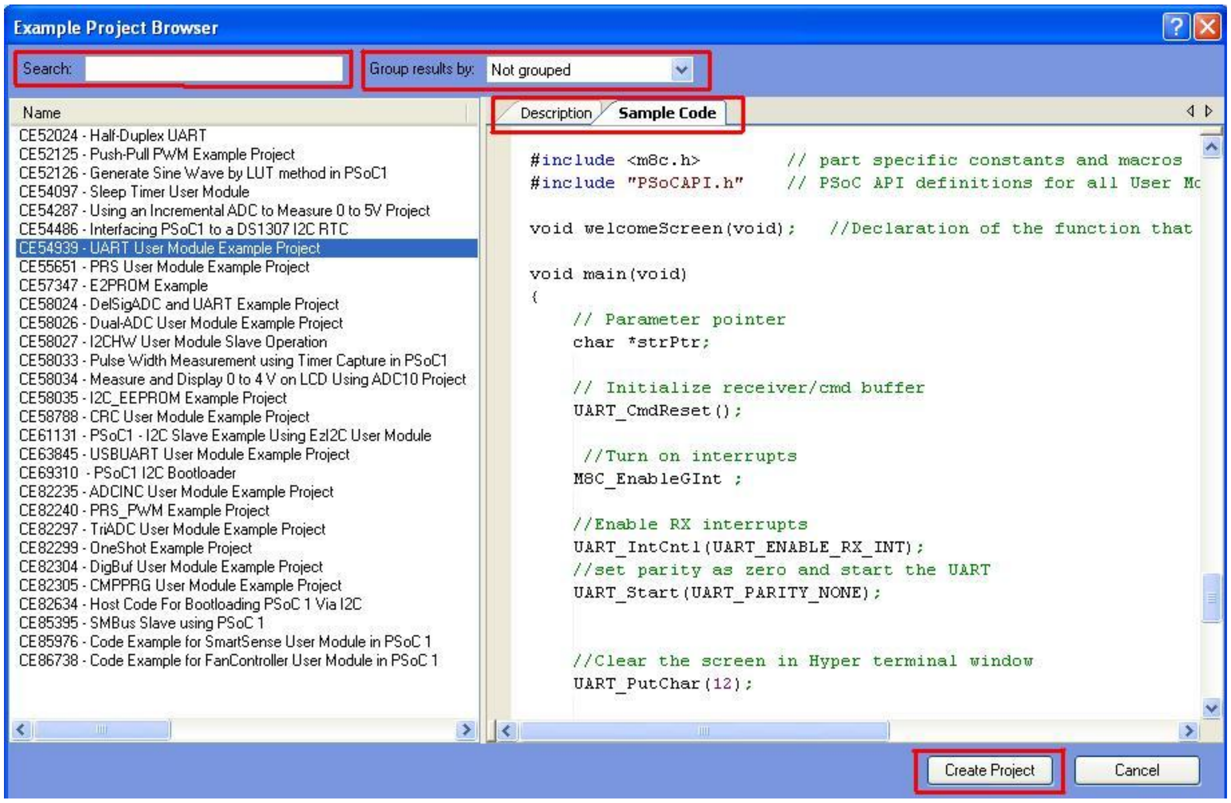
Proximity_CY3280_21x34
CY3280_21x34_SLM
CY3280_20x34_SLM
AN2141
HSSP_CY8Cxxxx
AN2272_F14

CE52125 - Push-Pull PWM Exar
CE54287 - Using an Incrementa
CE54939 - UART User Module E
CE82235 - ADCINC User Modul
CE63845 - USBUART User Moc
CE82240 - PRS_PWM Example
CE55651 - PRS User Module E

In the Example Projects Browser shown in Figure 3 you have the following options.

- Keyword search to filter the projects.
- Listing the projects based on Category.
- Review the datasheet for the selection (on the Description tab).
- Review the code example for the selection. You can copy and paste code from this window to your project, which can help speed up code development, or
- Create a new project (and a new workspace if needed) based on the selection. This can speed up your design process by starting you off with a complete, basic design. You can then adapt that design to your application.

Figure 3. Code Example Projects, with Sample Codes



CE52024 - Half-Duplex UART
CE52125 - Push-Pull PWM Example Project
CE52126 - Generate Sine Wave by LUT method in PSoC1
CE54097 - Sleep Timer User Module
CE54287 - Using an Incremental ADC to Measure 0 to 5V Project
CE54486 - Interfacing PSoC1 to a DS1307 I2C RTC
CE54939 - UART User Module Example Project
CE55651 - PRS User Module Example Project
CE57347 - E2PROM Example
CE58024 - DelSigADC and UART Example Project
CE58026 - Dual-ADC User Module Example Project
CE58027 - I2CHW User Module Slave Operation
CE58033 - Pulse Width Measurement using Timer Capture in PSoC1
CE58034 - Measure and Display 0 to 4 V on LCD Using ADC10 Project
CE58035 - I2C_EEPROM Example Project
CE58788 - CRC User Module Example Project
CE61131 - PSoC1 - I2C Slave Example Using EzI2C User Module
CE63845 - USBUART User Module Example Project
CE69310 - PSoC1 I2C Bootloader
CE82235 - ADCINC User Module Example Project
CE82240 - PRS_PWM Example Project
CE82297 - TriADC User Module Example Project
CE82299 - OneShot Example Project
CE82304 - DigBuf User Module Example Project
CE82305 - CMPPRG User Module Example Project
CE82634 - Host Code For Bootloading PSoC 1 Via I2C
CE85395 - SMBus Slave using PSoC 1
CE85976 - Code Example for SmartSense User Module in PSoC 1
CE86738 - Code Example for FanController User Module in PSoC 1

```
#include <m8c.h>        // part specific constants and macros
#include "PSoCAPI.h"    // PSoC API definitions for all User Mc

void welcomeScreen(void);   //Declaration of the function that

void main(void)
{
    // Parameter pointer
    char *strPtr;

    // Initialize receiver/cmd buffer
    UART_CmdReset();

     //Turn on interrupts
    M8C_EnableGInt ;

    //Enable RX interrupts
    UART_IntCntl(UART_ENABLE_RX_INT);
    //set parity as zero and start the UART
    UART_Start(UART_PARITY_NONE);


    //Clear the screen in Hyper terminal window
    UART_PutChar(12);
```

2.3 Technical Support

If you have any questions, our technical support team is happy to assist you. You can create a support request on the Cypress Technical Support page.

You can also use the following support resources if you need quick assistance.

- Self-help
- Local Sales Office Locations

3 Debugging Hardware and Setup

This section describes the hardware required to set up a debugging system with PSoC1 device. Included are an ICE, a debug pod, and the appropriate connectors. To understand the complete debugging hardware setup, see the video on AN73212 web page. The following sections discuss details of each element.

3.1 On-Chip-Debugger (OCD) Parts

Not all PSoC1 chips support debugging. But every PSoC1 device family has at least one version of the device that supports debugging. These debugging-enabled PSoC devices are called OCD parts. An OCD part for a device family can emulate any part number within the given family. For example, a CY8C27002-24PVXI OCD part can emulate any CY8C27xxx series part.

Some device families have multiple OCD parts that offer different packages, while others have only one OCD part for an entire family. Refer to the device datasheet to find the part number and part pinout for the OCD part for each PSoC1 device family.

3.2 In-Circuit Emulator (ICE)

The In-Circuit Emulator (ICE), or ICE-Cube, is a key element in a PSoC1 debugging system. The ICE manages all the emulation communication between the debugger software running on the PC (PSoC Designer) and the PSoC OCD chip. The ICE is shown in Figure 4.

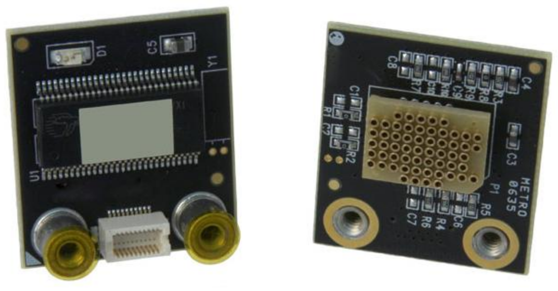
Figure 4. In-Circuit Emulator (ICE)

The ICE is available in the CY3215A-DK development kit, which comes with the ICE and all required accessories to enable debugging.

3.3 Debugging Pods

A debugging pod is another important element in a PSoC1 debugging system. In general, a pod is a small PCB that contains an on-chip-debugger (OCD) enabled PSoC1 chip, an ICE connector, and a connector that links the PCB to an existing board or development kit. Two primary types of debugging pods are available: CY3250 pod (see Figure 5) and CY3210 EvalPod (see Figure 6).

Figure 5. Example CY3250 Pod



The CY3250 pod is best suited for adding temporary debugging capability to a new or existing customer PCB. The CY3250 pod consists of an OCD PSoC part, an ICE connector, a connector that can be mounted to a board through a foot-kit, and a few passive components for the power supply.

For a list of available CY3250 pods, refer to PSoC® 1 Kit Selector Guide or the Emulation and Programming Accessories table in the device datasheet for the PSoC1 device being used.

The CY3210-EvalPod is best suited for prototyping with an evaluation kit or any other board with a 28-pin DIP connector. A CY3210-EvalPod is available for most PSoC1 device families. Each EvalPod comes with an OCD-enabled PSoC1 device, a 28-pin PDIP connector for board mounting, an RJ-45 connector for ICE connection, a 5-pin programming header and connectors for easy access to pins for prototyping. In general, EvalPods work best with the CY3210-PSoCEval1 development kit, bread-board, or any other board with a 28-pin PDIP connector.

Figure 6. Example CY3210 EvalPod

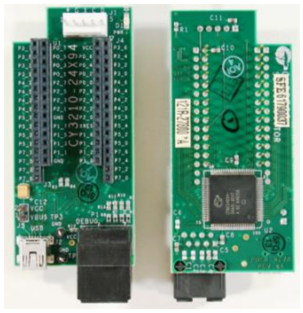


Table 1 lists available CY3210 EvalPods. Most common families have a CY3210 EvalPod available.

Table 1. CY3210-EvalPod List

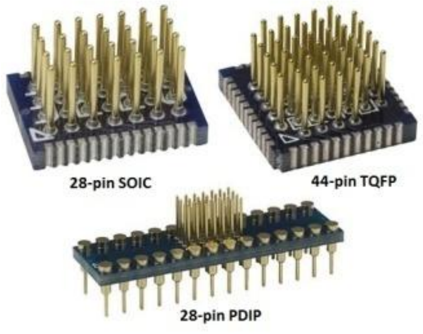
Part Family	EvalPod Part Number
CY8C21x23	CY3210-21x23
CY8C24x23	CY3210-24X23
CY8C24x94	CY3210-24x94
CY8C27xxx	CY3210-27x43
CY8C28xxx	CY3210-28xxx
CY8C29xxx	CY3210-29x66

As an alternative to using a pod, you can mount an OCD part and ICE connector directly to a new PCB. Direct mounting is useful for cases in which a pod may not fit on the PCB or if there is no pod available for the targeted PSoC1 device. For more information on building debugging capability directly onto a PCB, please refer to Appendix A – Adding an OCD Part to a PCB.

3.3.1 Feet

To properly mount CY3250 style pods on a board, a foot kit and mask are required. A foot kit allows a pod to be mounted to a specific package footprint, enabling a single pod to be mounted to almost any board. A foot kit, with a CY3250 pod and an ICE, allows an existing PCB to be modified to enable debugging. The foot kit with the appropriate footprint can be soldered to the existing PCB and the pod can be mounted on the foot. Figure 7 shows examples of three feet.

Figure 7. Example Feet



As shown in Figure 7, the bottom half of the foot adapts to the package footprint on the PCB. The figure shows a 28-pin SOIC, a 44-pin TQFP, and a 28-pin PDIP, but foot kits are available for all PSoC1 package sizes and types.

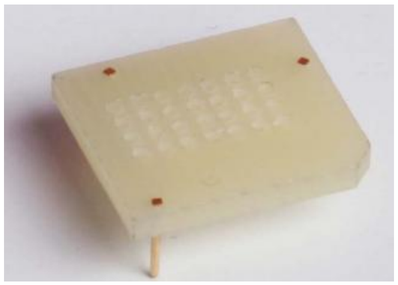
The top half of the foot kit provides connection to the appropriate pins of the pod. The number of pins varies by package, so each foot kit connects to the necessary pins of the OCD part. For example, an OCD part on a pod may come in a 56-QFN package; to emulate a part that has only 28 pins available, the foot kit will connect to only 28 of the 56 available pins.

The Emulation and Programming Accessories table in the device datasheet lists the appropriate foot kit for each PSoC device package.

3.3.1 Masks

A mask, which fits between the pod and the foot kit, aligns the pins from the foot kit to the appropriate holes in the pod and masks unused pins. Some pods have many connection points and some feet have few (an 8-pin package, for instance). Therefore, a mask ensures that the foot kit is aligned properly with the pod connector. An example mask is shown in Figure 8.

Figure 8. Example Mask



Not all feet and pod combinations require a mask, because some pods fit only into a foot kit in one orientation. For example, none of the QFN-based pods require a mask to connect to a QFN foot kit. Two mask sizes are common: a mask compatible with 28-pin feet (DIP, SOIC and SSOP packages), and a mask that works with any 8-pin or 20-pin foot kit. Feet with 44 or 48 pins do not require a mask.

If required, the appropriate mask is supplied with the CY3250 pod or foot kit and does not need to be purchased separately.

3.4 Debugging Hardware Setup

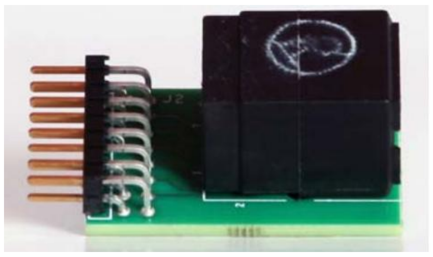
This section shows how to connect the various hardware pieces to build a functional debugging system. Depending on which pod, foot kit and mask are used, the configuration may vary slightly. Several common configurations are shown below.

3.4.1 ICE + CY3210 EvalPod

To connect an ICE with a CY3210 EvalPod, you need these pieces:

1. ICE
2. CY3210 EvalPod for the targeted PSoC1 family
3. RJ-45 ICE adapter, as shown in Figure 9, also known as a ‘backward compatibility adapter,’ because some legacy pods also used an RJ-45 connector.

Figure 9. RJ-45 ICE Adapter



4. Short RJ-45/Ethernet cable (12 inches or shorter), as supplied with the CY3215A-DK kit. A longer cable cannot be substituted because it would interfere with communication between the ICE and the pod.

Figure 10. RJ-45/Ethernet cable

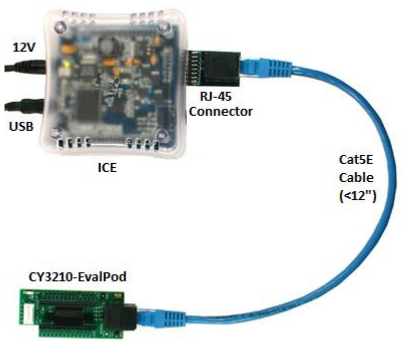
If a replacement cable is needed, a standard Cat5E cable that is 12 inches or shorter may be used.

Note: ICE communication is not standard Ethernet protocol. A custom communication protocol is used.

5. USB cable and 12V power supply (both are supplied with the ICE CY3215A-DK).

   The appropriate connections for the pieces listed above are shown in Figure 11. The EvalPod may optionally be plugged into a board with a 28-pin DIP connector.

Figure 11. ICE + CY3210 EvalPod Configuration



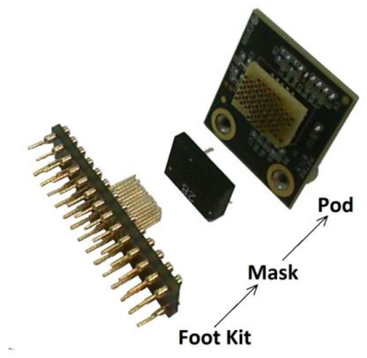
3.4.2 ICE + CY3250 Pod

To connect an ICE to a CY3250 pod, take the following steps:

1. Select a foot, which should match the pinout of the PSoC device in the target circuit.
2. If required, select a mask that matches the desired foot. Typically only 8-pin, 20-pin and 28-pin feet need a mask.
3. Insert the mask into the bottom of the pod, aligning the chamfered corners of the mask to the pin-1 triangle on the pod.
4. Insert the foot through the mask. If no mask is used, the foot connects directly to the base of the pod.

   Steps 2 through 4 are shown in Figure 12.

Figure 12. Foot Kit + Mask + Pod



5. Connect the assembled pod to the target board (if the foot-kit is not already attached to a board).
6. Connect the pod to the small end of the CY3250-Flex cable. Pod, cable and connector are shown in Figure 13. The completed connection is shown in Figure 14.

Figure 13. CY3250 Pod, Flex Cable and Connector

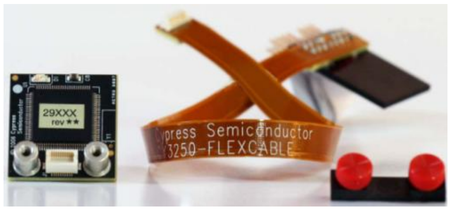
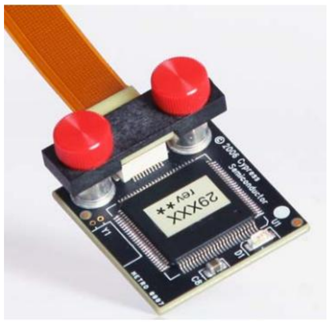
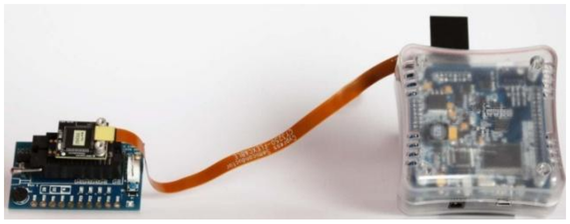


Figure 14. Flex Cable Connection



7. Plug the other end of the flex cable into the ICE as shown in Figure 15.

Figure 15. ICE Connected to CY3250 Pod



8. Finally, connect the ICE to the PC via a USB cable and power the ICE through the supplied 12V DC power supply.

   Table 2 shows the pin name and the pin description of the Pod connector used during debugging process.

Table 2. Pin Description of Pod connector

Pin Number	Pin Name	Pin Description
1	POD EXTRA3	Future use
2	GND	Ground
3	—	—
4	POD_OCDDE	POD_OCD even data I/O
5	GND	Ground
6	POD_OCDDO	POD_OCD odd data output
7	POD EXTRA1	Future use
8	POD_XRES	Reset signal (required only for Reset programming mode)

Pin Number	Pin Name	Pin Description
9	GND	Ground
10	POD_OCDHC	POD_OCD high speed clock output
11	GND	Ground
12	POD_OCDCC	POD_OCD CPU clock output
13	POD EXTRA4	Future use
14	PODVCC	Supply voltage
15	—	—
16	PODVCC	Supply voltage

3.4.3 ICE + OCD Device Mounted on Board

In some cases, instead of using a pod for debugging the OCD part is mounted directly on a target board. The appropriate debugger lines are brought out to a connector also mounted on the board. This is done in cases where a pod is either unavailable for a targeted PSoC1 device or it does not fit on the target PCB.

Placing an OCD device directly on the board sometimes may be a more cost-effective way to enable debugging. Several PSoC1 development boards use this method to enable debugging. Those boards are typically identified by an on-board RJ-45 connector.

OCD Interface Signals: The ICE contains an exact (emulated) copy of the M8C microcontroller. The ICE will start both microcontrollers (the M8C and its copy) at the same time and keep them running in sync. The M8C must be able to send IO data to the ICE fast enough to keep the M8C and ICE in sync. The ICE will communicate with the remote M8C by way of an eight-wire interface (RJ-45 connector). Table 3 shows the OCD interface signals used in debugging process.

Table 3. OCD Interface Signals

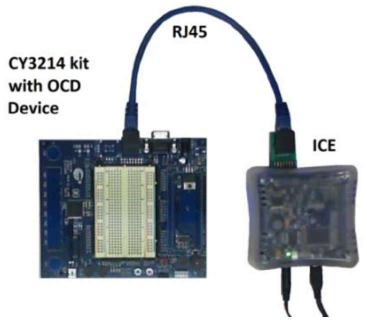
Pin Number	Direction (with respect to ICE)	Signal Name	Description
1	In	U_HCLK	24/48 MHz debug clock driven by remote M8C. This clock is used to drive the ICE/M8C communication state machines. This clock always runs at 24 MHz, unless the U_CCLK clock is running at 24 MHz and then it switches to 48 MHz.
2	Out	ICE_POD_GND	Ground wire to remote M8C.
3	Out	ICE_POD_RST	Active high reset signal to remote M8C.
4	Out	ICE_POD_GND	Ground wire to remote M8C.
5	In	U_CCLK	The internal M8C CPU clock.
6	In	U_D1_IRQ	One of two data lines used by the M8C to send IO data to the ICE. This line is also used to notify the ICE of pending interrupts. This line is only driven by the M8C.
7	InOut	U_D0_BRQ	One of two data lines used by the M8C to send IO data to the ICE. The ICE uses this line to convey halt requests.
8	Out	ICE_POD_PWR	Optional power supply to remote M8C.

a. RJ45 connector pin numbers.

For more information on how to add an OCD part directly to a board, refer to Appendix A – Adding an OCD Part to a PCB. The On-board OCD Debugger schematic is shown in Figure 29.

Connecting an ICE to a board with a mounted OCD device is similar to the process shown above to connect an ICE to a CY3210-EvalPod. The only difference is that the cable is connected to the target board RJ-45 connector, rather than to an EvalPod. Refer to Figure 16 for an example.

Figure 16. ICE + CY3214 with On-Board OCD Device



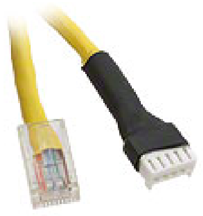
3.4.4 Using the ICE to Program a Device

The ICE also can program a PSoC device using the 5-pin in-system serial programming (ISSP) cable supplied with the CY3215A-DK.

Follow these steps:

1. Connect the ICE to the RJ-45 adapter.
2. Connect the ISSP cable to the RJ-45 adapter. The cable, typically yellow or black, has an RJ-45 connector at one end and a white 5-pin connector at the other end.

Figure 17. ICE ISSP Cable



3. Connect the 5-pin female connector to the 5-pin programming header on the target board. Figure 18 shows an ICE and target board configured for ISSP.

Figure 18. ICE Configured for Programming

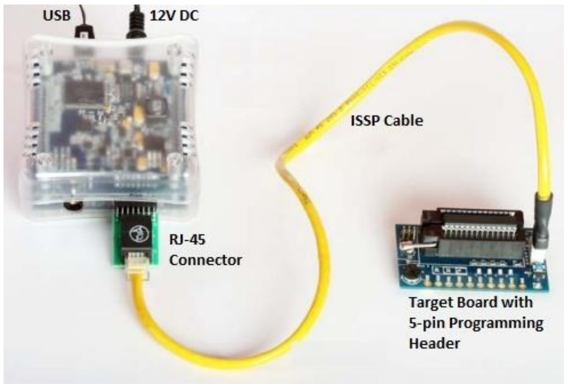
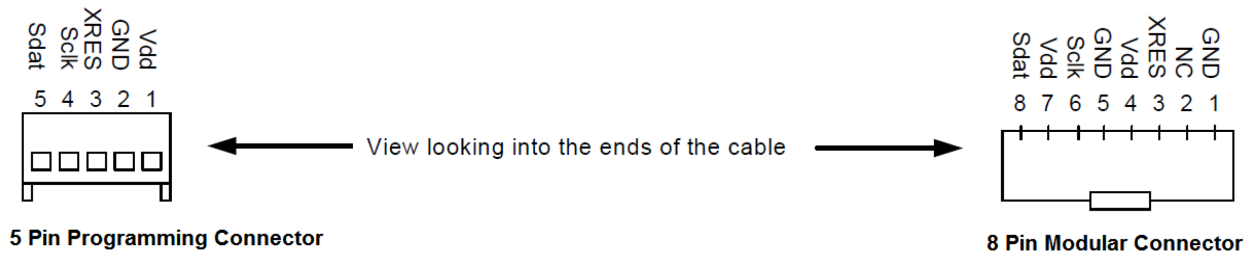


Table 4. 5-Pin Programmer Connector to 8-Pin Modular Connector

Signal	5-Pin Programmer Connector	8-Pin Modular Connector
V_{dd}	Pin 1	Pin 4 and Pin 7
GND	Pin 2	Pin 1 and Pin 5
XRES	Pin 3	Pin 3
Sclk	Pin 4	Pin 6
Sdat	Pin 5	Pin 8



Note: Debugging in this configuration is not possible. The 5-pin ISSP cable connects to the device's programming pins, which are different from the pins used for debugging.

4 Debugging Environment – PSoC Designer IDE

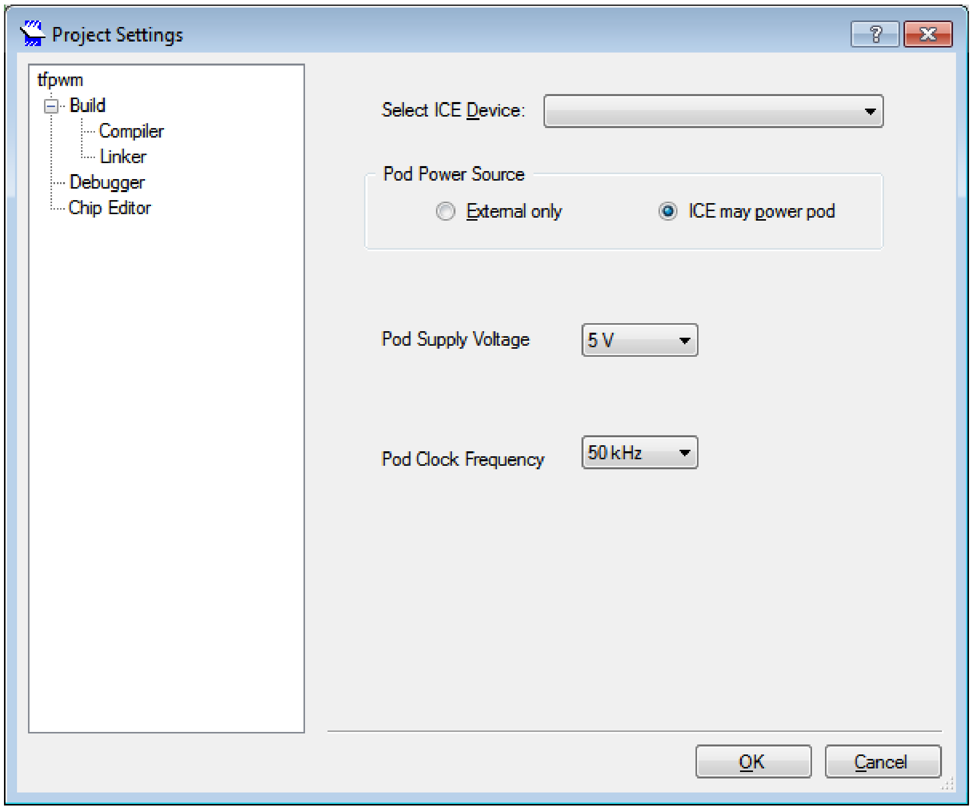
PSoC Designer includes a powerful set of code development and debugging tools. The debugging tools can be used with the ICE and any of the hardware configurations described in the Debugging Hardware Setup section of this application note.

The critical debugging elements within PSoC Designer are described briefly below. For additional information on each feature, refer to the Debugger section of the IDE User Guide. To download the IDE Guide click here or view it in **PSoC Designer > Help > Documentation > Designer Specific Documents > IDE User Guide.pdf**.

4.1 Starting the Debugger

To connect PSoC Designer to an ICE and begin debugging, ensure that the ICE and board are powered and the ICE is connected to the PC with a USB cable. The target board may be powered independently or via the ICE. To control how the target board is powered, go to the debugger settings within PSoC Designer. The debugger settings can be accessed through the **Project -> Settings -> Debugger** window, as shown in Figure 19.

Figure 19. Debugger Project Settings



Select ICE Device:

Pod Power Source: ( ) External only [✓ ICE may power pod]

Pod Supply Voltage: 5 V

Pod Clock Frequency: 50 kHz

The settings in Figure 19 allow you to select which ICE to connect to if multiple ICE devices are connected to a single PC. The settings also allow you to configure the ICE to power the pod. If the ICE powers the pod, the pod supply voltage must also be specified. The only available supply voltages are 5.0 V and 3.3 V.

After the ICE is connected to the PC and the appropriate settings are applied, the system is ready to debug. The PSoC Designer ICE toolbar provides controls to connect to the ICE; the functions for which are described in

Table 5.

Table 5. ICE Connection Controls

Icon	Function	Description
	Connect	Connects PSoC Designer to the attached ICE Cube.
	Download	Downloads compiled code to the ICE and OCD part.
	Refresh M8C Views	Updates the memory and variable displays shown in PSoC Designer.
	Get Next Trace Data	Adds 64 lines of information about code execution to the trace window. See the Trace section for more information.

If you have trouble connecting and downloading the program to the ICE, please refer to the Troubleshooting section in this document.

4.2 Debugging Controls

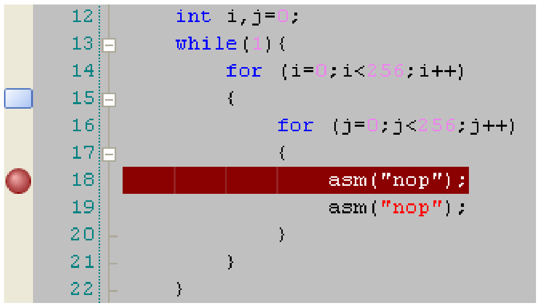
PSoC Designer has a number of commands to control the execution of code on the attached ICE and OCD PSoC. The debugger supports several standard methods, shown in Table 6, for controlling the program flow.

Table 6. Basic Debugging Controls

Icon	Function	Description
	Go	Starts debugger.
	Run to Cursor	Creates a temporary (invisible) breakpoint at the current cursor location in the source code and runs the application to that point.
	Halt	Stops the Debugger.
	Reset	Resets the device to a PC value of ‘0’ and restarts the debugger.
	Step Into	Steps into the next statement.
	Step Out	Steps out of the current function.
	Step Over	Steps over the next statement.
	Step ASM	Executes one line of assembly code. If the current line of code is C code, the list file is opened and a single line of assembly is executed.
	Execute Program	Connects to the ICE, downloads the program and starts to run it.

Breakpoints are another debugging tool of PSoC Designer and the ICE. A breakpoint is a marker in code that indicates execution should pause when the marked line is reached. You can add breakpoints by left-clicking in the margin next to the line of code or by right-clicking in the code editor on the appropriate line and selecting "Insert Breakpoint." A line of code with a breakpoint inserted is shown in Figure 20. Also shown is a "bookmark" (line 15), which you can add by right clicking in the margin next to the line of code. Although bookmarks have no impact on code execution, they are useful for marking points of interest in code.

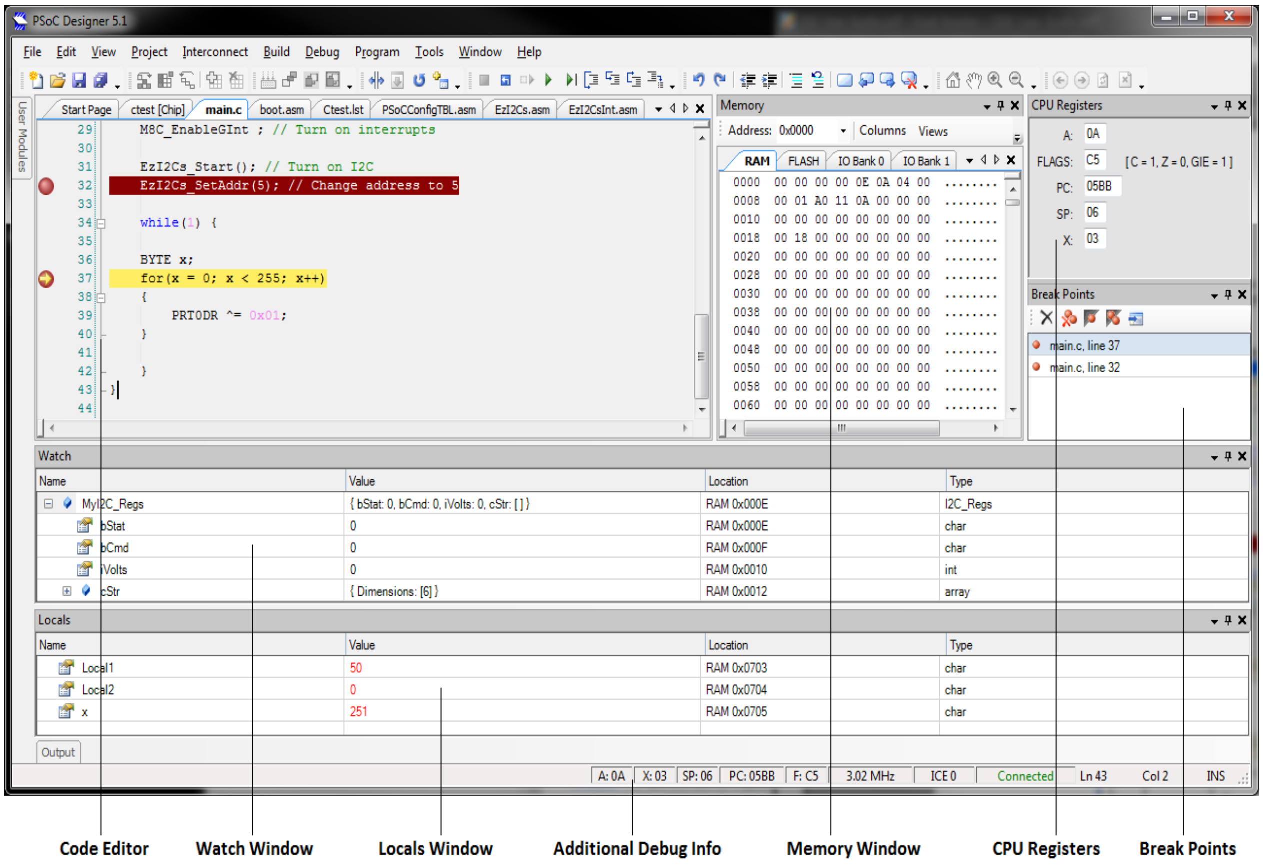
Figure 20. Bookmark and Breakpoint Example



```
12        int i,j=0;
13        while(1){
14            for (i=0;i<256;i++)
15            {
16                for (j=0;j<256;j++)
17                {
18                    asm("nop");
19                    asm("nop");
20                }
21            }
22        }
```

Figure 21 shows many of the debug windows available during a debugging session in PSoC Designer. Each window shown is optional and can be moved and resized to your liking. A brief description of each window follows, and additional information on each window can be found in the IDE User Guide. To download the IDE Guide click here, or view it in **PSoC Designer > Help > Documentation > Designer Specific Documents > IDE User Guide.pdf**.

Figure 21. PSoC Designer Debug Windows


PSoC Designer 5.1
File Edit View Project Interconnect Build Debug Program Tools Window Help
User Modules
Start Page | ctest [Chip] | main.c | boot.asm | Ctest.lst | PSoCConfigTBL.asm | EzI2Cs.asm | EzI2CsInt.asm
Memory
Address: 0x0000 Columns Views
RAM | FLASH | IO Bank 0 | IO Bank 1
CPU Registers
Break Points
Watch
Locals
Output
Code Editor
Watch Window
Locals Window
Additional Debug Info
Memory Window
CPU Registers
Break Points

```
29      M8C_EnableGInt ; // Turn on interrupts
30
31      EzI2Cs_Start(); // Turn on I2C
32      EzI2Cs_SetAddr(5); // Change address to 5
33
34      while(1) {
35
36      BYTE x;
37      for(x = 0; x < 255; x++)
38      {
39          PRT0DR ^= 0x01;
40      }
41
42      }
43  }
44
```

```
0000   00 00 00 00 0E 0A 04 00   ........
0008   00 01 A0 11 0A 00 00 00   ........
0010   00 00 00 00 00 00 00 00   ........
0018   00 18 00 00 00 00 00 00   ........
0020   00 00 00 00 00 00 00 00   ........
0028   00 00 00 00 00 00 00 00   ........
0030   00 00 00 00 00 00 00 00   ........
0038   00 00 00 00 00 00 00 00   ........
0040   00 00 00 00 00 00 00 00   ........
0048   00 00 00 00 00 00 00 00   ........
0050   00 00 00 00 00 00 00 00   ........
0058   00 00 00 00 00 00 00 00   ........
0060   00 00 00 00 00 00 00 00   ........
```

A: 0A
FLAGS: C5 [C = 1, Z = 0, GIE = 1]
PC: 05BB
SP: 06
X: 03

main.c, line 37
main.c, line 32

Name	Value	Location	Type
MyI2C_Regs	{ bStat: 0, bCmd: 0, iVolts: 0, cStr: [ ] }	RAM 0x000E	I2C_Regs
bStat	0	RAM 0x000E	char
bCmd	0	RAM 0x000F	char
iVolts	0	RAM 0x0010	int
cStr	{ Dimensions: [6] }	RAM 0x0012	array

Name	Value	Location	Type
Local1	50	RAM 0x0703	char
Local2	0	RAM 0x0704	char
x	251	RAM 0x0705	char

A: 0A | X: 03 | SP: 06 | PC: 05BB | F: C5 | 3.02 MHz | ICE 0 | Connected | Ln 43 | Col 2 | INS

Code Editor Window: The code editor window displays the code during the debug session. The window also displays where breakpoints are located and where code execution has halted (if the program is not running). Code edits cannot be made during debugging. If edits are made to the code while the debugger is active, PSoC Designer will exit debug mode and allow the changes to be made to the code.

Watch Window: The watch window displays any watch variables the user has added during the debug session. To add a variable to the watch window, right-click in the code editor window during a debug session, and then click **Add Watch**.

Locals Window: The locals window displays any local variables that are in scope when the program is halted. The local variables in the current scope will automatically be added.

Additional Debug Information: At the bottom of the screen, PSoC Designer always displays useful debug information. These include:

- the current values for the accumulator (A)
- X register (X)
- stack pointer (SP)
- program counter (PC)
- flag register (F)

Information about the ICE connection speed and connection status also are displayed.

Memory Window: The memory window allows a user to inspect the various memory areas on the chip while halted. Each area (RAM, FLASH, IO Bank 0, IO Bank 1) has its own tab within the memory debug window. RAM, IO Bank 0 and IO Bank 1 can be edited during an active debug session.

CPU Registers Window: The CPU registers debug window display the various internal CPU registers for the PSoC device. The registers displayed are the accumulator (A), flags register, program counter (PC), stack pointer (SP) and X register. Values displayed in the CPU register window are editable during an active debug session.

Breakpoints Window: The breakpoints window displays all of the placed breakpoints in the project, including the file and line number for each. Individual or all breakpoints can be deleted, disabled, or enabled from this window.

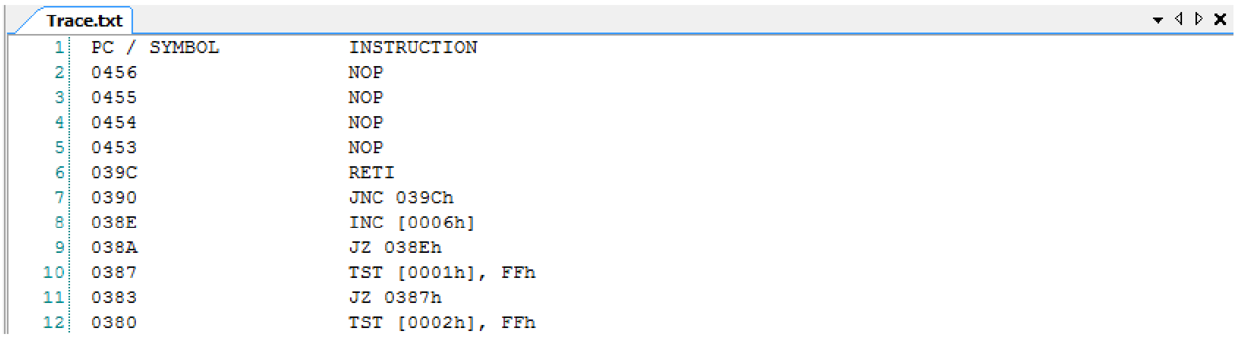
4.3 Trace

A trace window in PSoC Designer and the ICE debugger lets a user track and log device activity instruction by instruction. The window displays a continuous, configurable listing of internal operations of the device. Each time program execution is started, the trace buffer is cleared. When the buffer becomes full, it continues to operate and overwrite old data.

Three trace modes are available and described below.

Trace Mode 1 - PC Only – Lists the program counter (PC) and the assembly instruction for each instruction executed by the PSoC.

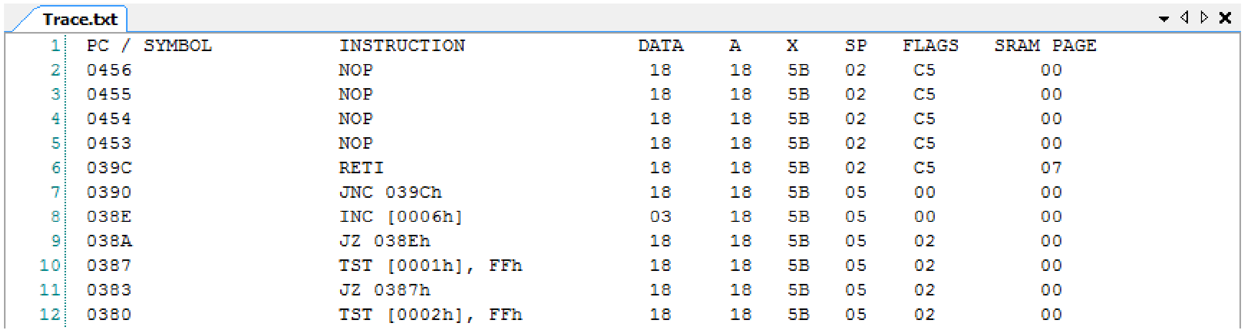
Figure 22. Trace Mode 1 - PC Only



```
PC / SYMBOL            INSTRUCTION
0456                   NOP
0455                   NOP
0454                   NOP
0453                   NOP
039C                   RETI
0390                   JNC 039Ch
038E                   INC [0006h]
038A                   JZ 038Eh
0387                   TST [0001h], FFh
0383                   JZ 0387h
0380                   TST [0002h], FFh
```

Trace Mode 2 - PC/Registers – Lists the PC, instruction, data, accumulator (A), X register, stack pointer (SP), flag register, and SRAM page.

Figure 23. Trace Mode 2 - PC/Registers

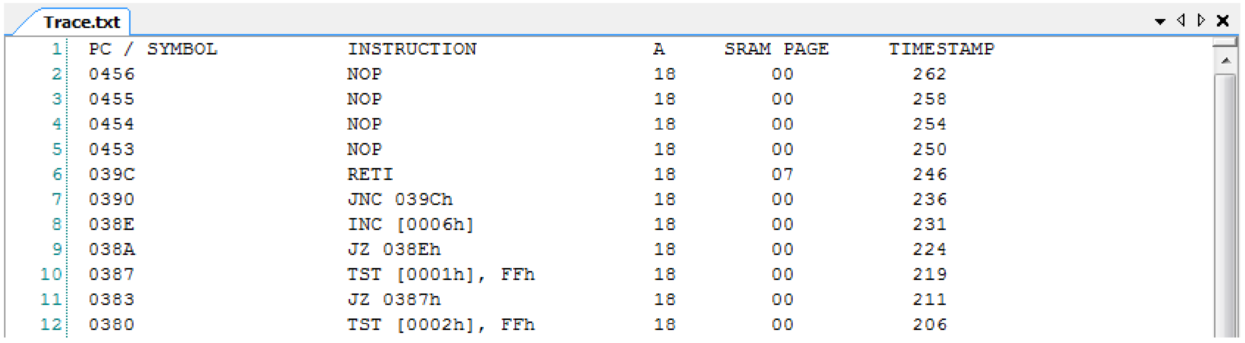


Trace.txt

```
PC / SYMBOL         INSTRUCTION            DATA   A    X    SP   FLAGS   SRAM PAGE
0456                NOP                     18    18   5B   02    C5        00
0455                NOP                     18    18   5B   02    C5        00
0454                NOP                     18    18   5B   02    C5        00
0453                NOP                     18    18   5B   02    C5        00
039C                RETI                    18    18   5B   02    C5        07
0390                JNC 039Ch               18    18   5B   05    00        00
038E                INC [0006h]             03    18   5B   05    00        00
038A                JZ 038Eh                18    18   5B   05    02        00
0387                TST [0001h], FFh        18    18   5B   05    02        00
0383                JZ 0387h                18    18   5B   05    02        00
0380                TST [0002h], FFh        18    18   5B   05    02        00
```

Trace Mode 3 - PC/Timestamp – Lists the PC, instruction, accumulator (A), SRAM page and timestamp. The timestamp, which keeps a running tally of CPU cycles executed by the program, is an efficient way to count the CPU cycles that a section of code requires to execute.

Figure 24. Trace Mode 3 - PC/Timestamp



Trace.txt

```
PC / SYMBOL         INSTRUCTION           A     SRAM PAGE     TIMESTAMP
0456                NOP                   18       00            262
0455                NOP                   18       00            258
0454                NOP                   18       00            254
0453                NOP                   18       00            250
039C                RETI                  18       07            246
0390                JNC 039Ch             18       00            236
038E                INC [0006h]           18       00            231
038A                JZ 038Eh              18       00            224
0387                TST [0001h], FFh      18       00            219
0383                JZ 0387h              18       00            211
0380                TST [0002h], FFh      18       00            206
```

The trace buffer size is 256 kB,, which allows 128k trace instructions to be tracked in trace mode 1 and 32k trace instructions to be tracked in trace modes 2 and 3.

The trace window is available through the **Debug->Windows->Trace** menu. The trace mode can be changed using the **Debug->Trace Mode** menu. Once the trace window is open, it will load the latest 64 trace entries into the trace window when the program is halted. To load 64 additional entries, use the 'Get Next Trace Data' toolbar or menu option. Alternatively, the 'Get all Trace Data' menu option lets you load all available trace data.

4.4 Events Viewer

The Events Viewer lets you define conditions or sequences of conditions that, when met, trigger either breakpoints or traces. As a result, you can inspect and debug more complex sequences of code that would otherwise not be possible with standard debug tools.

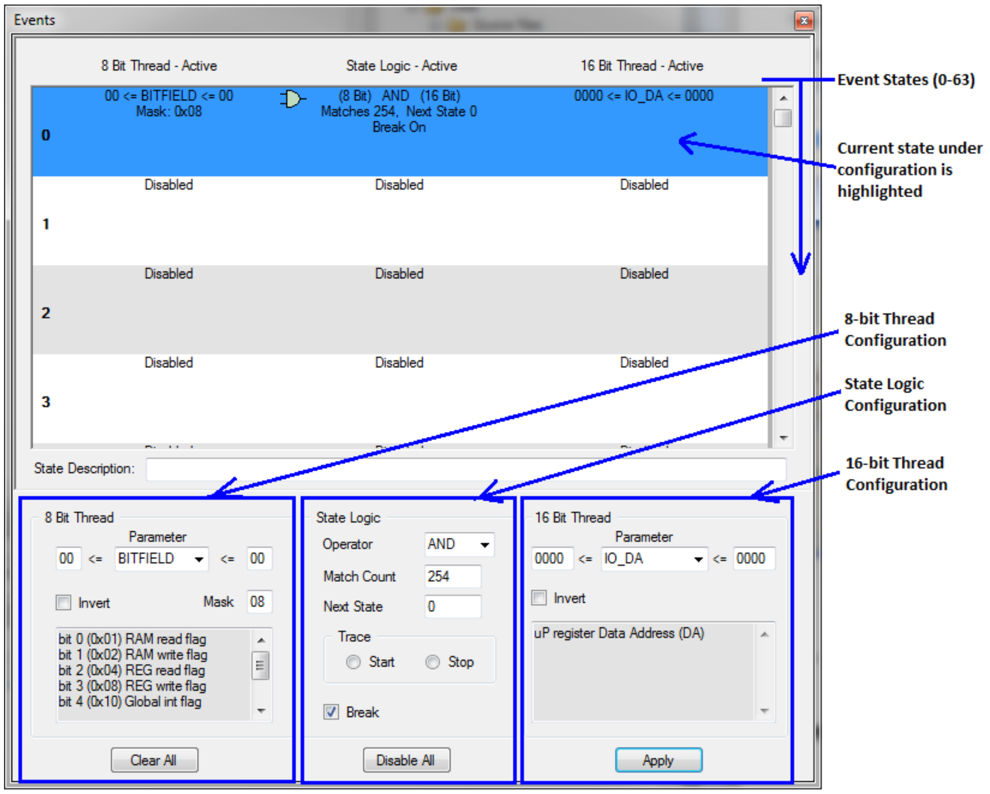
Events can be configured to be useful in many debug situations. Setting up an event can be compared to a standard if-then statement. First, a comparison is made. If that comparison is evaluated as true, then an action can be taken. The action can be to set a breakpoint, start or stop the trace buffer, or move on to another chained if-then statement. As many as 64 statements can be defined.

The 'if' portion of the statement is made up of an 8-bit or 16-bit thread. Each thread allows a comparison to be made on several available 8-bit or 16-bit sources such as a RAM address, PC value, or stack value.

To add flexibility to the 'if' statement, you can use 8-bit and 16-bit comparisons independently or combined via a logic function (OR, AND, NOR, NAND). In addition, you can invert the output logic of each individual 8-bit or 16-bit thread to better support checking of active low, or cleared data.

The Events Debug Window, shown in Figure 25, is accessible in PSoC Designer under the **Debug > Windows > Events** menu. The window has a section for selecting and displaying each of the available 64 event states. To configure a specific event state, simply select it in the upper portion of the window. Set the configuration parameters of the state set in the lower portion. Details on each available setting are provided in the sections that follow.

Figure 25. Events Debug Window


Event States (0-63)
Current state under configuration is highlighted
8-bit Thread Configuration
State Logic Configuration
16-bit Thread Configuration

Events

	8 Bit Thread - Active	State Logic - Active	16 Bit Thread - Active
0	00 <= BITFIELD <= 00 Mask: 0x08	(8 Bit) AND (16 Bit) Matches 254, Next State 0 Break On	0000 <= IO_DA <= 0000
1	Disabled	Disabled	Disabled
2	Disabled	Disabled	Disabled
3	Disabled	Disabled	Disabled

State Description:

8 Bit Thread
Parameter
00 <= BITFIELD <= 00
Invert Mask 08
bit 0 (0x01) RAM read flag
bit 1 (0x02) RAM write flag
bit 2 (0x04) REG read flag
bit 3 (0x08) REG write flag
bit 4 (0x10) Global int flag

State Logic
Operator AND
Match Count 254
Next State 0
Trace: Start / Stop
[✓ Break]

16 Bit Thread
Parameter
0000 <= IO_DA <= 0000
Invert
uP register Data Address (DA)

Clear All | Disable All | Apply

The available 8-bit thread, 16-bit thread, logical comparison and output options for events are discussed below.

4.4.1 8-bit Thread Options

The 8-bit thread offers several ways to evaluate 8-bit wide parameters within the PSoC. Comparisons can be made on specific values or on a range of values between 0x00 and 0xFF.

Invert: This option lets you invert the outcome of the 8-bit parameter comparison.

Mask: The mask field allows specific bits to be masked before the parameter comparison is made on the input data. This option is useful when you are interested in only particular bits of the parameter selection value.

Parameters: The parameter drop-down menu allows you to select the field to be used in the 8-bit comparison. The available 8-bit parameter comparison options are shown below.

NONE: Disables the 8-bit thread comparison.

A: Accumulator comparison. Evaluates the data in the accumulator.

IO_DA: Register address comparison. Evaluates the data address in the register memory space.

IO_DB: Register data comparison. Evaluates the data on the data bus in the register memory space.

IR: Instruction register comparison. Evaluates the instruction opcode in the Instruction register.

MEM_DA: RAM address comparison. Evaluates an address in the RAM memory space.

MEM_DB: RAM data comparison. Evaluates the data on the data bus in the RAM memory space.

SP: Stack pointer comparison. Evaluates the current address of the stack pointer (not the data to which the SP is pointing).

X: X register comparison. Evaluates the current value in the X register.

BITFIELD: Enables specific flag checks on the chip's internal operations, such as when a write or read occurs or when the zero or carry flag has been set. The available BITFIELD checks are shown below. Multiple flags can be checked simultaneously.

Bit 0 (0x01): RAM read flag

Bit 1 (0x02): RAM write flag

Bit 2 (0x04): Register read flag

Bit 3 (0x08): Register write flag

Bit 4 (0x10): Global interrupt flag

Bit 5 (0x20): Zero flag

Bit 6 (0x40): Carry flag

Bit 7 (0x80): Extended IO flag

BITFIELD compare options are typically combined with an additional compare (or sequence of compares). For example, bit 1 (RAM write flag) is often used with a RAM address and/or RAM data compare (in a 16-bit thread) to see when a specific value is written to a RAM location. The RAM address and RAM data compares will perform these tasks on their own, but will continue to evaluate to true even after the initial write occurs (which may not be desirable depending on the debug situation).

The bits within the BITFIELD are active low. To check for specific bits in the BITFIELD:

- Use the Mask field to check for the bit(s) of interest. For example, to check for the zero flag set the Mask value to 0x20. Doing so masks the unwanted bits so they will not be evaluated by the parameter comparison.
- Set the high and low parameter comparison values to 0. Because the mask has already screened for the bits of interest, the parameter comparison no longer is needed. A value of 00 in the high and low comparisons allows any bits that pass through the mask to be evaluated by the comparison (because BITFIELD bits are active low).

  For specific examples demonstrating the BITFIELD comparison parameter, refer to Example 1: Break on Write of Specific Address and Example 2: Break on Write of Specific Value and Address.

4.4.2 16-Bit Thread Options

The 16-bit thread offers several ways to evaluate 16-bit-wide parameters within the PSoC. You can make comparisons on specific values or a range of values between 0x0000 and 0xFFFF. Note that some of the available comparison options in the 16-bit threads are only 8-bits wide and will evaluate values between 0x00 and 0xFF.

Invert: The invert option lets you invert the outcome of the 16-bit parameter comparison.

Parameter: The parameter drop-down offers many of the comparisons available in an 8-bit thread but includes the ability to make comparisons on a few additional 16-bit wide parameters. Some of the 16-bit comparisons allow two 8-bit parameters to be checked simultaneously in a single comparison. The available 16-bit comparison options are shown below.

NONE: Selecting this parameter disables the 16-bit thread comparison.

A[1]*:* Accumulator comparison offers the same functionality as 8-bit thread version.

IO_DA[1]*:* Register address comparison has the same functionality as 8-bit thread version.

IO_DA_DB: Combined register address and data compare. The MSB 8-bits entered will be used for a register address comparison (similar to IO_DA); the LSB 8-bits entered will be used for a register data comparison (similar to IO_DB).

IO_DB[1]*:* Register data comparison has the same functionality as 8-bit thread version.

IR[1]*:* Instruction register comparison offers the same functionality as 8-bit thread version.

IR_SP: Combined Instruction register and stack pointer comparison. The MSB 8-bits perform an instruction register comparison; the LSB 8-bits perform a stack pointer comparison.

MEM_DA[1]: RAM address comparison, same functionality as the 8-bit thread version.

MEM_DA_DB: Combined RAM address and RAM data comparison. The MSB 8-bits perform a RAM address comparison (similar to MEM_DA); the LSB 8-bits perform a RAM data comparison (similar to MEM_DB).

MEM_DB[1]: RAM data comparison has the same functionality as the 8-bit thread version.

PC16: Program counter comparison. This parameter lets you evaluate the current value in the program counter register.

SP[1]: Stack pointer comparison offers the same functionality as the 8-bit thread version.

X[1]: X register comparison has the same functionality as the 8-bit thread version.

X_A: Combined X register and accumulator comparison. The MSB 8-bits perform a comparison on the X register; the LSB 8-bits perform a comparison on the accumulator.

Note 1: Only the LSB 8-bits entered into the 16-bit-wide comparison entry fields are evaluated for the A, IO_DA, IO_DB, IR, MEM_DA, MEM_DB, SP and X comparisons. The page pointer registers will not automatically be checked for a greater-than-8-bit comparison on RAM or register address comparisons.

4.4.3 State Logic

Operator: The operator field allows the 8-bit and 16-bit threads to be logically combined if both are enabled. Available logical operators are AND, OR, NAND, NOR. If either the 8-bit or 16-bit threads are disabled, the operator is ignored.

Match Count: Match count allows the user to specify how many times the event should occur before enabling the trace or breakpoint or moving onto another event state. The match count can be between 1 and 32,767. If more than 32,767 matches are needed, you can chain together multiple event states.

Next State: The next state parameter allows multiple event states to be chained together, to create more complex and flexible event sequences. When the current event state, including the specified number of matches is met, the next state will be executed. When you do not want state chaining, the next state should be set to the current state number.

Not all states are evaluated simultaneously. State 0 is always first. If state 0 is disabled or evaluates false, no further states will be evaluated. As such all events, or chains of events, must start with state 0 even when no chaining is desired.

Trace: The trace buffer may be started or stopped as possible outcomes from a successful event state match. This is useful when only a specific section of code is intended to be traced.

Break: The break checkbox allows you to specify if you would like this event to induce a breakpoint when the event conditions are met.

4.4.4 Notes about Events

- After breaking on a successful event condition, PSoC Designer will halt 2 instructions after the event occurs. The reason for the stoppage is that the ICE evaluation of event parameters lags the actual pod execution by 2 instruction cycles. Sometimes, that behavior will not be obvious in C debugging, because multiple lines of assembly may be required for each line of C code.
- Changes in the events window cannot be applied to a running device. When changes are made to an event or series of events, you must click the apply button when the device is halted to ensure the ICE recognizes the updated event.
- When using the BITFIELD compare parameter in an 8-bit Thread, you should do two things: Set the high and low compare values to 0x00 and check the appropriate BITFIELD parameter using the Mask field. For a sample setup and use of the BITFIELD parameter refer to Example 1: Break on Write of Specific Address.
- When an event configuration evaluates to true and induces a break, the configuration will continue to evaluate to true (and continue to cause breaks) until the state of the chip changes to state that no longer causes the event to evaluate to true. For example, if an event is configured to break when a specific value is loaded into the accumulator, but the instructions following the condition do not modify the accumulator, breaks will continue to occur (1 instruction at a time) until the accumulator is changed.

4.4.5 Common Uses for Events

Events can be used in many debugging situations, but common potential uses include:

- Find when a register or RAM address is read or written (see Event Examples 1 and 2 below).
- Find a stack overflow (see Example 3: Stack Overflow).
- Detect a jump or call out of a function.
- Trace a specific range of code.
- Measure interrupt latency.
- Break the nth time a line of code executes (match count, see Example 4 below).
- Break on carry flag status.
- Wait for a certain number of instructions.
- Break on specific data in the accumulator.

Details on how to set up and use the events tool for a few common use cases are shown below.

4.4.6 Example 1: Break on Write of Specific Address

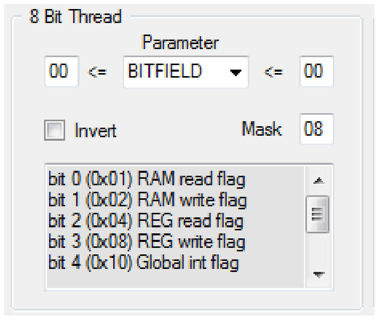
In some situations, it is valuable to understand when a specific address is written to.

For example, you are having an intermittent problem with an I2C slave in which a master periodically reads an incorrect value from the slave. You may not know when the I2C data register is being loaded with this incorrect value. An event can be used to break when the I2C output data register is written to.

To accomplish this, use an 8-bit thread to check when a write on a register address occurs. This check can be combined with a 16-bit thread that checks the I2C data register address. When a write occurs on that address, a breakpoint occurs.

To break on a write to a specific address, configure the following parameters:

1. Set the 8-bit thread parameters.



8 Bit Thread

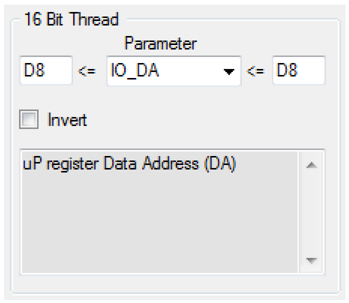
Parameter: 00 <= BITFIELD <= 00

Invert ☐ Mask 08

bit 0 (0x01) RAM read flag
bit 1 (0x02) RAM write flag
bit 2 (0x04) REG read flag
bit 3 (0x08) REG write flag
bit 4 (0x10) Global int flag

   a. Parameter = BITFIELD. This setting lets you make a variety of checks, including the ‘register write’ check, used in this example.
   b. Set the high and low compare value for the parameter to ‘00’, which allows all bits to be checked because the BITFIELD is active low. The mask value is used to check the specific bit of interest.
   c. Set the mask value to 0x08, allowing the BITFIELD comparison to evaluate to true whenever a register write is performed.

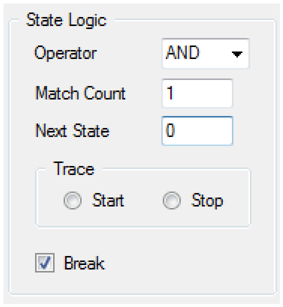
2. Set the 16-bit thread parameters.


16 Bit Thread
Parameter
D8 <= IO_DA <= D8
☐ Invert
uP register Data Address (DA)

a. Parameter = IO_DA. This setting enables an event to occur when a specific register address is on the bus.

b. Set the high and low compare values to ‘D8’. This is the data register for the I2C block. This register, or the registers for any other block, can be found in the user module datasheet or Technical Reference Manual.

3. Set the State Logic fields.


State Logic
Operator: AND
Match Count: 1
Next State: 0
Trace: ○ Start ○ Stop
☑ Break

a. Set the Operator field to ‘AND’. This configures the event state so that the conditions defined in the 8-bit thread AND the 16-bit thread both must be met before the event evaluates to true. In this case, the event will evaluate to true only when a write occurs on the I2C data register (reads will be ignored).

b. Set the match count to ‘1’. This event only needs to occur once before a breakpoint is triggered.

c. Set the Next State to ‘0’. In this case, only one event state is needed to evaluate the desired conditions so no state chaining is required.

d. Leave the ‘Start’ and ‘Stop’ options for Trace unconfigured. No trace is used in this example.

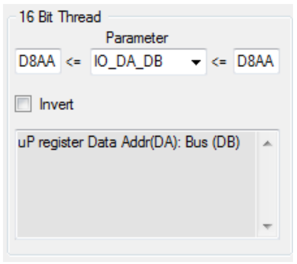
e. Enable the ‘Break’ checkbox. This will cause a breakpoint to occur whenever the conditions defined for the entire event state evaluate to true.

4.4.7 Example 2: Break on Write of Specific Value and Address

To expand on Event Example 1, a designer may also want to know when a specific value is written to a register. The event shown in example 1 will break every time a value is written to the I2C data register, but it may be more beneficial to break only when a specific incorrect value is written to the register.

To add this functionality, you must slightly modify the 16-bit thread. The other portions of the event state need to remain the same, as specified in example 1.

1. Set the 16-bit thread parameters.


16 Bit Thread
Parameter
D8AA <= IO_DA_DB <= D8AA
☐ Invert
uP register Data Addr(DA): Bus (DB)

a. Parameter = IO_DA_DB. This setting enables an event to occur when a specific value is written to a specific register address on the bus.

Set the high and low compare values to ‘D8AA’. Doing so will cause a break to occur whenever 0xAA is written to the I2C data register (0xD8). Note that a range of values, not just 0xAA, could be checked by changing the high or low compare values.

4.4.8 Example 3: Stack Overflow

Another useful scenario for the events viewer is to check for stack overflow, which can cause unexpected behavior and at times can be difficult to debug. Fortunately, the events viewer makes it easy to detect stack overflow and monitor stack usage.

To monitor a program for stack overflow with events, you must configure an 8-bit thread to watch the stack pointer. To break when a stack overflow is about to occur, the following parameters should be configured.

1. Set the 8-bit thread parameters:
   a. Parameter = ‘SP’. This setting allows the stack pointer value to be monitored.
   b. Set the low compare value to FD and the high compare value to FF. Doing so will induce a break whenever the stack pointer reaches FD.
   c. Set the mask value to FF. No bits will be masked in this setup.
2. Set the 16-bit parameter value to NONE to ensure the 16-bit thread is disabled.
3. Set the State Logic fields.
   a. The operator selection is not important, because the 16-bit thread is disabled. The operator selection is ignored when the 16-bit thread is disabled.
   b. Set the Match Count to 1. This event needs to occur only once before the breakpoint is triggered.
   c. Set the Next State to 0. In this case, only one event state is needed to evaluate the desired conditions, eliminating the need for state chaining.
   d. Leave the ‘Start’ and ‘Stop’ options for Trace unconfigured. No trace is used in this example, but users may find the trace feature useful when trying to determine what caused the stack overflow.
   e. Enable the ‘Break’ checkbox. This will cause a breakpoint to occur whenever the conditions defined for the entire event state evaluate to true.

For additional information on stack overflow, including a method for checking for stack overflow without a debugger and techniques for avoiding stack overflow, refer to Appendix B – Stack Overflow.

4.4.9 Example 4: Break After X Occurrences of Event

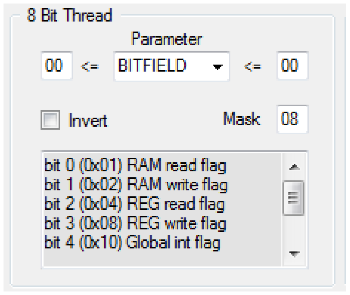
This example describes how to set a breakpoint after an event condition has evaluated true multiple times. This action helps a developer exercise finer control over when the program stops execution.

For this example, the events viewer is set up to break on the 254th time through a for-loop in the following code:

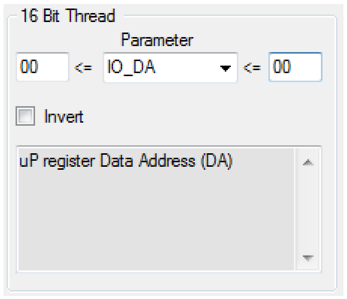
```
BYTE x;
for(x = 0; x < 255; x++)
{
   PRT0DR ^= 0x01;
}
```

The configuration for this event state is similar to the state shown in example 1. The BITFIELD parameter is used to detect a register write. This, combined with a 16-bit thread check on the PRT0DR register, allows the event count to increment each time through the loop. To configure this event state, make the following changes to the event:

1. Set the 8-bit thread parameters. These are the same settings as used in example 1. Please refer to example 1 for more detailed descriptions of each of these settings.


8 Bit Thread
Parameter
00 <= BITFIELD <= 00
Invert   Mask 08
bit 0 (0x01) RAM read flag
bit 1 (0x02) RAM write flag
bit 2 (0x04) REG read flag
bit 3 (0x08) REG write flag
bit 4 (0x10) Global int flag

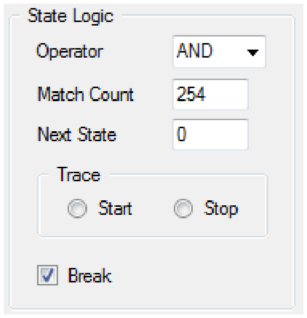
2. Set the 16-bit thread parameters. These settings are similar to the ones used in example 1, except the address compare has been changed to 00 (PRT0DR address).



00 <= IO_DA <= 00

uP register Data Address (DA)

3. Set the State Logic fields. The only difference from example 1 is that the match count has been changed to ‘254’. This will cause the event to evaluate to true and trigger a breakpoint after going through the loop 254 times.



Operator: AND
Match Count: 254
Next State: 0
Break: ✓

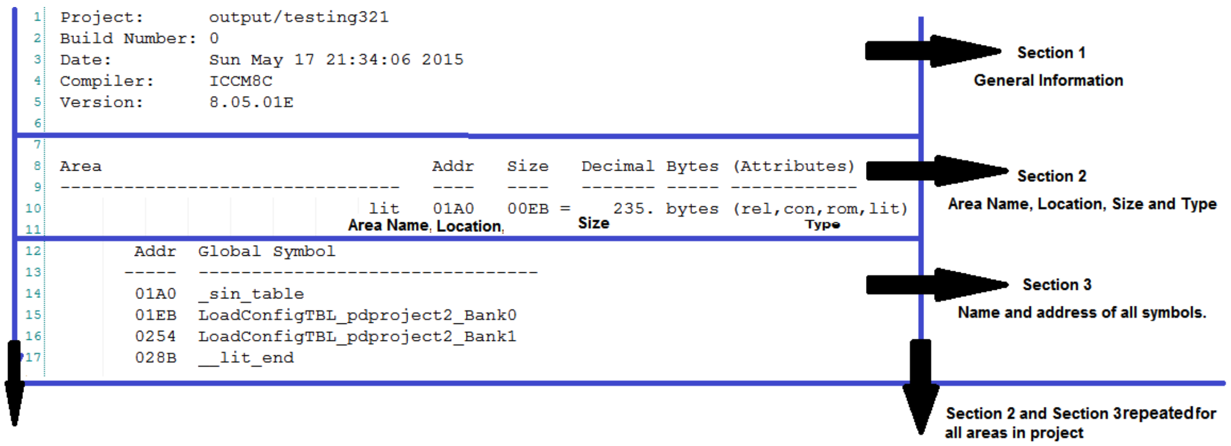
4.5 Map File (.mp)

A map file, which is generated during the build process, contains global symbol addresses and information on the areas defined in memory. The map file is a useful debugging tool when trying to determine where a variable or piece of code is located in memory, or when trying to determine how much memory a portion of code occupies.

Any variable, function, or label that is not global will not show up in the map file.

Each section of the map file is described in Figure 26. The layout and descriptions are specific to the ImageCraft C compiler and linker V7.0.5, and may vary slightly between compiler versions. The map file is available in the ‘Output Files’ folder within the Workspace Explorer in PSoC Designer.

Figure 26. Map File with Comments



```
Project:       output/testing321
Build Number:  0
Date:          Sun May 17 21:34:06 2015
Compiler:      ICCM8C
Version:       8.05.01E


Area                              Addr   Size   Decimal Bytes (Attributes)
--------------------------------  ----   ----   ------- ----- ------------
                             lit  01A0   00EB =    235. bytes (rel,con,rom,lit)

       Addr  Global Symbol
      -----  --------------------------------
       01A0  _sin_table
       01EB  LoadConfigTBL_pdproject2_Bank0
       0254  LoadConfigTBL_pdproject2_Bank1
       028B  __lit_end
```

Area Name, Location,
Size
Type
Section 1
General Information
Section 2
Area Name, Location, Size and Type
Section 3
Name and address of all symbols.
Section 2 and Section 3 repeated for all areas in project

Section 1: General Information

The beginning of the map file (lines 1-5) lists general information about the project including the project name, the date, the timestamp for the last build and the compiler version.

Section 2: Area Name, Location, Size and Type

The next section (lines 8-10) is a header for the next area of memory. This section shows the name of the area, the location (memory address), size, and type of area.

Area Name: This is assigned when the area is declared. An area is a section of memory where the compiler will place code and data. Users may specify their own memory areas, but the ImageCraft Compiler uses several predefined areas:

> top: Contains the interrupt vectors and boot.asm code. This area is placed at address 0 in ROM.
>
> lit: Contains constant data, placed in ROM.
>
> idata: Stores the initialization values for the global data.
>
> text: Contains user code.
>
> psoc_config: Contains functions used to configure the device (and functions for dynamically reconfiguring the device).
>
> usermodules: Contains the user module API routines.

Area Location: This is the starting address of the area. This is a ROM address for areas defined in ROM and a RAM address for those defined in RAM.

Area Size: This is the total size of the area, provided in both hex and decimal. The end of the area will be the area location plus the area size.

Type of Area: These are the defined attributes for the given area. They define what kind of data is placed in the area and how the linker will place the area. The following is a complete list of the valid keywords that you can use to define the type of area:

> **REL –** Relocatable area. This keyword allows the linker to relocate the area to the address of its choosing. Relocatable areas do not have a predefined location, their address is determined during the linking stage of the build.
>
> **ABS –** Absolute area, or non-relocatable area. This keyword means the area address is defined in code and will not be moved by the linker.

An area can be either relocatable or absolute, but not both.

> **CON –** Concatenated area. This keyword specifies that areas will follow each other sequentially in memory.
>
> **OVR –** Overlay area. This keyword specifies that multiple overlay areas can have the same start address and can occupy the same memory space. This is done only with RAM areas.

An area can be either concatenated or overlay, but not both.

> **RAM –** Specifies that the data is stored in RAM and is used only for variable storage.
>
> **ROM –** Specifies that the data is stored in flash. Both code and constant data can be stored in ROM areas.

An area can be either RAM or ROM, but not both.

In this example, the area shown is the ‘lit’, or literal data area. This area stores any constant data used in the project. The total size is 235 bytes, and the area type, ‘rel, con, rom’, refers to a relocatable, concatenated area in the code (ROM) memory space.

Section 3: Name and address of all symbols.

This section of the map file describes the contents of the area described and defined in section 2. The address and name of each symbol placed in the area are listed sequentially. Because each symbol is placed sequentially, the size of each element can be determined by comparing the address of the given symbol to the address of the next symbol.

The address listed corresponds to the memory type for the given area (RAM areas will be RAM addresses and ROM areas will be ROM addresses).

Not all symbols in the area will necessarily occupy memory. Global symbol labels will also appear in the list. For example, in Figure 26 the '__lit_end' symbol simply indicates where the lit area ends (address 0x028B) but does not physically occupy any memory.

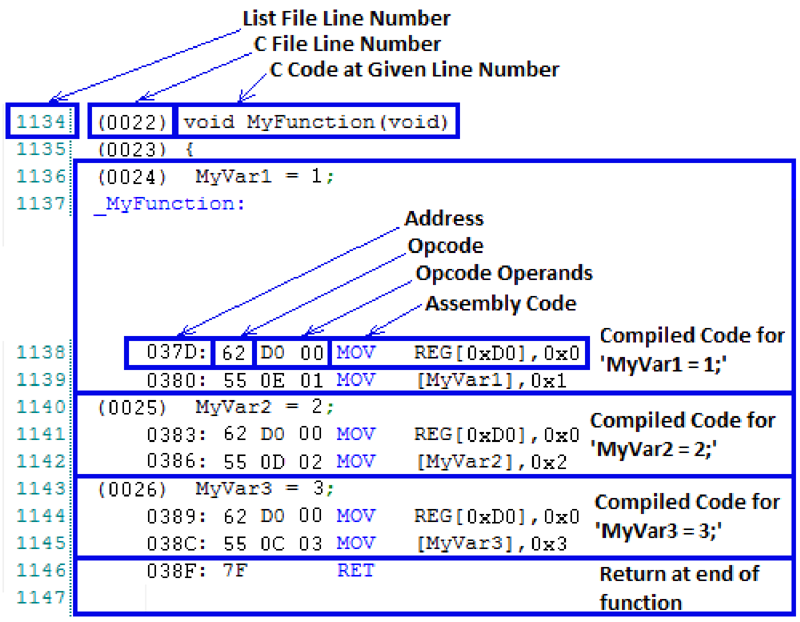
4.6 List File (.lst)

Another useful file generated by PSoC Designer is the list (.lst) file, which contains the assembly code generated by the C compiler. Refer to the list file when debugging a function that is not operating as expected or when trying to interface a portion of C code with a portion of assembly code. The 'Step ASM' function of the debugger also will direct a user into the list file.

Figure 27 shows an excerpt from a list file, highlighting critical information. For each line in a C source file the list file relists the line of C code and sequentially lists all of the compiled assembly required for the given line. The list file is organized starting at address 0x00 of flash and lists all compiled code sequentially until the end of memory. As such, in large projects the file can become long. The search function is an easy way to find a particular function or piece of code within a large list file.

Each line of assembly also lists the code address for the instruction and the specific opcode and opcode operands for the assembly instruction. For additional information on assembly opcodes, refer to the Assembly Language User Guide (**PSoC Designer > Documentation > Compiler and Programming Documents > Assembly Language User Guide**).

Figure 27. List File with Comments


List File Line Number
C File Line Number
C Code at Given Line Number
Address
Opcode
Opcode Operands
Assembly Code
Compiled Code for 'MyVar1 = 1;'
Compiled Code for 'MyVar2 = 2;'
Compiled Code for 'MyVar3 = 3;'
Return at end of function

```
1134  (0022) void MyFunction(void)
1135  (0023) {
1136  (0024)   MyVar1 = 1;
1137  _MyFunction:
1138        037D: 62 D0 00 MOV   REG[0xD0],0x0
1139        0380: 55 0E 01 MOV   [MyVar1],0x1
1140  (0025)   MyVar2 = 2;
1141        0383: 62 D0 00 MOV   REG[0xD0],0x0
1142        0386: 55 0D 02 MOV   [MyVar2],0x2
1143  (0026)   MyVar3 = 3;
1144        0389: 62 D0 00 MOV   REG[0xD0],0x0
1145        038C: 55 0C 03 MOV   [MyVar3],0x3
1146        038F: 7F       RET
1147
```

This excerpt of the list file shows 'MyFunction', which assigns constant values to three global variables (MyVar1, MyVar2, MyVar3). As shown in the figure, each C assignment compiles into two lines of assembly code; a line to set the page register and a line to assign the value to the variable.

5 PSoC1 Debugging Tips and Tricks

This section documents some unique aspects to keep in mind when debugging a PSoC1 device.

5.1 Hardware Runs While 'Stopped'

The ICE can emulate and control the state of code execution within a PSoC device, but the ICE does not have explicit control over the digital and analog portions of the device. As such, whenever the debugger is halted, the other hardware elements continue to run. The hardware values reported by the debugger at a breakpoint are a snapshot of the value of these registers at the time of the halt.

This behavior is most obvious when working with counters and timers or user modules that use counters and timers, such as several ADC user modules. After the debugger halts execution of the CPU, the counter continues to run. Therefore, when the program is restarted, the first value reported by the timer/counter/pwm (and any ADCs using a counter), may be incorrect. After the initial incorrect value is reported the timer/counter/pwm (and any ADCs using a timer) will resynchronize and start to report the correct values again.

If the debugging depends on the accuracy of the ADC reading, there are two workarounds available to ensure the value reported by the debugger for the ADC is correct:

1. Flash Write Debugging As the program is running, dynamically set a breakpoint after the instruction that reads the ADC result, by left-clicking on the left margin of the code window (left side of the Line number). When the program halts, the value read from the ADC will be correct. To get another reading, clear the breakpoint, run the program and dynamically place the breakpoint again.
2. If the selected ADC supports single sample mode (such as an incremental ADC), another option is available. Instead of running the ADC in continuous sampling mode, run it in single sample mode. In single sample mode the counter and timer are stopped after each conversion, which ensures the value returned by the ADC is correct (as long as there is not a breakpoint between the start of the conversion and when the result is returned). Note this method does not apply to Delta-sigma based ADCs.

5.2 Flash Write Debugging

You can use the ICE and the debugger when developing a project that includes internal flash writes. Examples include a bootloader or other applications that save data to flash. When using the ICE debugger with flash writes, consider the following:

- To allow the ICE to emulate the flash write, the flash block being written to must be unprotected. Unprotected flash is the only security setting that allows external devices to write to flash. Even though the ICE is emulating an internal flash write, the ICE is still technically an external device requesting a flash update within the PSoC device.
- Emulated flash writes using the ICE take significantly longer than a flash write on a standalone PSoC device. The amount of communication required between the ICE and the PSoC device during a flash write takes time. Write times with the ICE vary, but they generally will be 10 to 20 times as long as the typical write times defined in the PSoC device datasheet.
- Do not attempt to execute a system supervisor call (SSC) instruction without using the `SSC_Action (OpCode)` macro in m8ssc.inc (included in every PSoC Designer project). The instructions leading up to a flash write, or any other SSC instruction, must follow a specific sequence so that the ICE can detect and emulate the SSC instruction. The `SSC_Action` macro executes the proper sequence of instructions to enable emulation during SSC operations. User modules and Cypress-provided libraries (such as the Flashblock write library) use the correct SSC macro.
- The E2PROM user module and the flashblock API library flash write routines are provided to users as pre-compiled libraries. The source is neither supplied nor available for the debugger to 'step into' during debugging. However, the functions will still execute correctly when 'stepped over'.

5.3 Using the Debugger with Sleep

When debugging a project that uses sleep, consider the following:

- Breakpoints cannot be placed on the line following an M8C_Sleep instruction. Sleep performs a pre-fetch of the next instruction before going to sleep. As a result, the ICE will become out of sync with the PSoC and disconnect if a breakpoint is encountered immediately after the sleep instruction.
- The debugger does not allow single-stepping into or over a sleep instruction; doing so causes the ICE to disconnect from the device. To avoid this, set a breakpoint 2 lines after the sleep instruction and allow the debugger to run freely into the breakpoint, rather than explicitly stepping over the sleep instruction.
- If the debugger is manually halted while the connected PSoC1 device is sleeping, the ICE will disconnect. To avoid this, projects using sleep should always halt a running program using a breakpoint or an event, rather than directly clicking the halt command.

6 Alternative Debugging Options

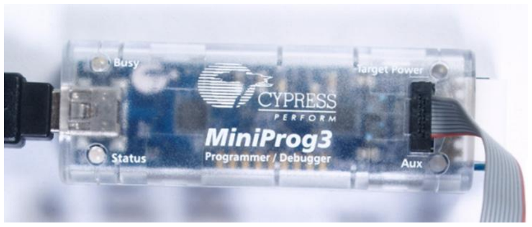
This section covers alternative debugging tools and techniques. These alternatives lack the feature set of a standard debugging setup, but they are a cost-effective option when a standard ICE debugging setup is not available or practical. Additionally, these alternatives work well in situations that require real-time debugging support by allowing debugging without physically stopping the operation of the device.

6.1 Debugging with an I2C to USB Bridge

An I2C to USB bridge allows you to test, tune and debug hardware and software of a PSoC application by bridging the USB port from a PC to an I2C user module in the PSoC. The bridge acts as an I2C host; issuing commands from the USB interface to any I2C slaves on the bus.

Cypress offers the I2C to USB bridge as CY8CKIT-002 PSoC MiniProg3 Program and Debug Kit. The MiniProg3 has built-in I2C to USB bridge functionality and can program all PSoC1, 3, and 5 devices. The MiniProg3 is shown in Figure 28.

Figure 28. MiniProg3 - I2C to USB Bridges



An I2C to USB bridge is useful for debugging real-time applications or in situations in which an ICE and OCD setup cannot be used. Neither an ICE nor an OCD-enabled PSoC are required to debug in this manner.

The typical use case involves the following elements:

- An I2C to USB Bridge.
- Target PSoC device with an EZI2C slave user module running that is connected to either P1[5] and P1[7] or P1[0] and P1[1].
- PC with Bridge Control Panel software running to issue commands to the bridge.

The I2C to USB bridge is then connected between the host PC and the target PSoC device acting as an I2C slave. The Bridge Control Panel is used to periodically query the PSoC device for information. The polling rate and amount of data transferred are completely configurable and can help you view real-time debugging information from a running PSoC device.

Common I2C to USB bridge debug use cases:

- Streaming real-time capsense button tuning data. The Bridge Control Panel also supports real time graphing of incoming I2C data. In this use case, the PSoC program regularly loads the button information into the I2C array and the I2C bridge periodically queries that data.
- Watching a set of predefined “watch” variables. In this use case, the PSoC program will regularly copy any variables of interest into the I2C array, and the I2C bridge periodically queries and watches this data.
- Testing the functionality of an I2C slave. The I2C to USB bridge acts as an I2C master and can test the functionality of any I2C slave on the bus.
- Testing the functionality of an I2C-based bootloader. Using an I2C to USB bridge is a common method for communicating with an I2C-based bootloader. PSoC Designer supports generating download files that the Bridge Control Panel can use directly for I2C bootloading operations.

For additional documentation on the bridge kits and associated documentation on how to use them, refer to CY8CKIT-002 PSoC MiniProg3 Program and Debug.

6.2 Debugging with a UART Interface

Debugging with a UART interface is a common strategy for any embedded processor. Fortunately, adding a UART interface to any PSOC1 device is easy, and many PSoC1 development kits have an RS-232 transceiver available to use.

In addition, you can view information transmitted by a UART device on almost any PC. Serial ports are becoming less common, but USB to UART bridge devices are widely available. PC software to view the data, such as Hyperterminal, is also widely available.

Common use cases for UART debugging are similar those listed for the I2C to USB bridge (above). The API provided with the UART user module make it easy to pass debug information to and from the PSoC device. API choices include PutString, PutChar, GetString, and GetChar.

For more information on configuring and using a UART on PSoC1, please refer to the UART user module datasheet available within PSoC Designer.

6.3 Pin Toggles

Although basic, pin toggles are a good debug tool, particularly in situations requiring real-time debugging. Pin toggles are frequently used in the following situations:

- Timing the length of a section of code by encasing the section of interest with two pin toggles. This technique is often used to time the execution time of interrupt service routines (ISRs).
- Checking to see if a section of code is being reached. While this task also can be done with a breakpoint, sometimes it helps to make this check without interfering with code execution.

7 Troubleshooting

This section lists common problems encountered while using the PSoC1 debugger and discusses potential resolutions.

Issue	Possible Resolutions
ICE Cannot Detect or Connect, or is Incompatible with the Pod	The pod does not match the selected device for the project. Each pod supports a range of devices within its family; refer to the PSoC device datasheet to ensure the pod or OCD part being used is compatible with the specific part number selected within the project.
	Ensure that the PSoC device is properly powered. The PSoC may be powered externally or via the ICE. The Project > Settings > Debugger setting (as shown in Figure 17) must match the power method chosen.
	ICE is not powered. ICE should be powered by a 12-V/1-A DC power supply. If a lower voltage supply is used, the ICE may light up but will not operate correctly.
	Pod may not be connected correctly. Refer to Debugging Hardware Setup section for common debug hardware configurations.
	Ensure that only a single instance of PSoC Designer or PSoC Programmer is open and connected to the target device.
	An obsolete version of the pod may be connected to the ICE. Ensure a current revision of the pod is used.
ICE Disconnects During Debug Session	ICE may not be powered properly. Ensure the ICE is powered by a 12V/1A DC supply. A different power supply may allow a device to connect but may fail after debugging begins.
	Check to see if any flash writes or other SSC instructions are occurring in the code. If the flash writes occur, refer to the Flash Write Debugging section for information on how to use the ICE debugger with flash writes.
	Check to see if any sleep operations occur in the code. If the sleep operations occur, refer to the Flash Write Debugging section for information on how to use the ICE with a project using sleep.
	Avoid using the phase locked loop (PLL) while debugging. To avoid this issue, debug with the PLL off and all external crystal hardware disconnected from the PSoC.
	Avoid using an external crystal oscillator (ECO) while debugging. To avoid this issue, debug with the ECO turned off and the internal main oscillator (IMO) turned on.
	Verify connections between the PC, ICE and PSoC are correct and tight. Refer to the Debugging Hardware Setup section for common debug hardware configurations.
Invalid Memory Reference Occurs During Debug Session	Invalid Memory References (IMRs) occur for the same reasons that can cause the ICE to disconnect during a debug session. See Flash Write Debugging section for suggestions on how to resolve this issue.
The Selected ICE Port Cannot be Found	The USB cable may be detached from either the ICE or the PC. Ensure the USB cable is connected properly and wait a few seconds before trying to connect again.
	The ICE may not be powered properly. Ensure it is powered by a 12V/1A DC power supply.
	Ensure only a single instance of PSoC Designer or PSoC Programmer is open and connected to the target device.
Program Execution Halts at Unexpected Locations	The low-voltage detect (LVD) interrupt might be triggered, an action that would halt the program at address 0x04 in boot.asm (the LVD interrupt vector) or at the LVD interrupt service routine (if defined). In this case: - Check the PSoC power supply. If the voltage to the PSoC fluctuates below the LVD trip voltage, set in the global parameter settings. The LVD interrupt will occur. - Check to ensure the ICE is correctly powered. The ICE should be powered with a 12-V/1-A DC power supply; anything less may cause unexpected behavior. - If the ICE is powering the target device or board, ensure the target does not require more than 250 mA of current.
	Ensure that stack overflow is not occurring. Stack overflow can corrupt the program counter (PC), if the PC value loaded on the stack during a function call is overwritten. This can cause program execution to reach unexpected locations in code. Refer to Appendix B – Stack Overflow in this application note for tips on detecting and preventing stack overflow.

Issue	Possible Resolutions
USB Hub Power is Exceeded	If the board is being powered via the ICE, the board may require more current than the ICE (and the USB hub) can supply. Use a direct external power supply (instead of supplying power with the ICE) or a USB port closer to the primary PC USB port.
No Events are Ever Detected	Ensure the events have been properly applied. The 'apply' button within the events window must be clicked when the debugger is halted in order for the event state(s) to be recognized by the ICE. Applying event states while the debugger is running or disconnected will not apply the event information to the ICE.
	Ensure the 'break' checkbox is selected for the event(s) of interest.
If All Else Fails	If possible, try to use a second set of hardware: - First, exchange the ICE with a different (preferably new) ICE to see if that solves the problem. If that works, the problem might lie with the ICE unit. - Exchange the pod or board with the PSoC unit for a new or different set of hardware. If this resolves the problem, there may be an issue with the pod, PCB, or ICE connector.
	Uninstall PSoC Programmer and PSoC Designer and reinstall the latest versions of each tool. Connection problems can arise if drivers or other debugging-related tools built into the software are not installed correctly. Reinstalling will ensure the necessary tools are installed correctly with the latest bug fixes.

8 Summary

Cypress offers many hardware and software tools to help you debug a PSoC1 project. This application note should give you the confidence to understand what hardware, software, and debugging techniques to which you have access for debugging any project or problem.

About the Author

Name:	Dan Sweet
Title:	Applications Engineer Sr.

A Appendix A: Adding an OCD Part to a PCB

Adding an on-chip debugger (OCD) enabled PSoC1 device directly to a PCB layout is a viable debugging solution for situations in which a foot kit and pod do not fit on a target PCB or a pod is unavailable for a given part number.

Adding an OCD part directly to a PCB may offer the following advantages over a standard debugging pod:

- The part cost is lower, because a foot kit and full pod assembly are not required.
- It's easier to create multiple boards with debugging capability than it is to buy a separate pod for each board that needs debugging.
- Less PCB space is needed for the PSoC chip because no foot kit is required. Board space for a connector is required, but it may be as far as 4 inches away from the OCD part.

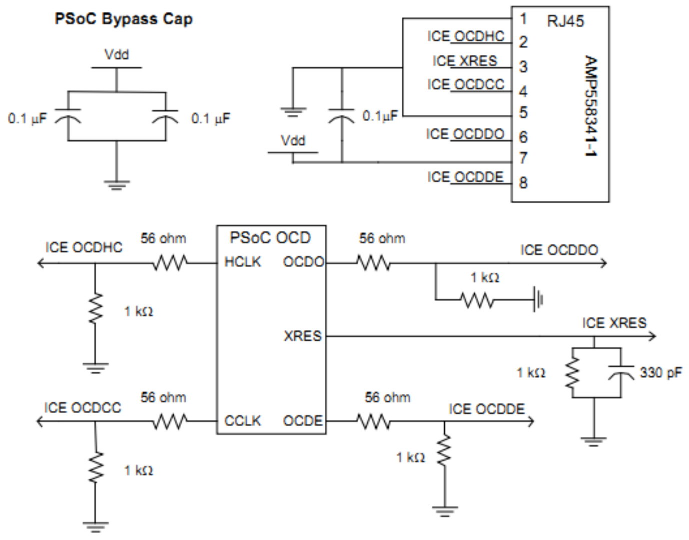
The components required to add an OCD part to a PCB are listed in Table 7.

Table 7. Components Required for an On-board OCD Debugger

Description	Quantity	Notes
RJ-45 Connector	1	AMP/Tyco Electronics Part Number 5557785-1 DigiKey Part Number A31457-ND
PSoC OCD Part	1	See On-Chip-Debugger (OCD) Parts.
56 Ω Resistor	4	1/16 W, 5%
1 kΩ Resistor	4	1/16 W, 1%
330 pF Capacitor	1	5-V Ceramic NPO
0.1-µF Capacitor	3	5-V Ceramic Y5V

The required schematic for the on-board OCD debugger is shown in Figure 29.

Figure 29. On-board OCD Debugger Schematic



All series resistors are used as termination, for impedance matching on the signal lines.

Bypass capacitors are included to filter out AC noise from critical elements in the circuit.

The trace length for OCDHC, OCDCC, OCDDO, and OCDDE should be restricted to less than 4 inches.

B Appendix B: Stack Overflow

B.1 Checking for Stack Overflow without an ICE

Earlier, this application note discussed a method for detecting stack overflow with the ICE and events (Example 3: Stack Overflow). While this method is an excellent way to capture and break on a stack overflow, it cannot work on a part without debugging capability.

Fortunately, stack overflow can be detected via firmware, too. One such method is to pre-fill the end of the stack page with a known value. If the stack reaches the end of the stack page and overwrites the pre-filled value with a new value, then there is a potential stack overflow issue.

To set a pre-fill value in the stack page, use the following C code:

```
char* StackPtr = (char*) 0x7FF;

*StackPtr = 0xAA;
```

The given code creates a pointer to the final RAM location in the stack page (in this case page 7) and then puts a pre-fill value of 0xAA in the location. PSoC1 devices with a single page of RAM should use a pre-fill address of 0x0FF. The third digit should always match the stack page used for the project.

The following code then can be used to check if the stack filled to this location at any point during code execution.

```
if (*StackPtr != 0xAA)
{
        //Stack overflow
}
```

The stack should have the pre-fill value applied early in the code initialization sequence before the stack grows too large. The pre-fill value can be checked at any point after that. It works best to have this in a section of code that runs regularly, such as the main loop. If the stack reaches the end of the page at any point during program operation, the check will detect it.

Note that if the fill value (in this example '0xAA') happens to be the same value that is pushed onto the stack at the pre-fill location, then the overflow will not be detected. If this is a concern, the test can be run multiple times with different pre-fill values or more than one stack location can be pre-filled and checked. Also note that the StackPtr location can be changed to any stack page or location within a stack page. That allows you to determine how full the stack is for a given project. For example, the StackPtr location can be placed at 0x780 or 0x7C0 to see if the stack on page 7 is reaching 50 percent or 75 percent full.

B.2 Methods to Avoid Stack Overflow

The following sections detail different methods of writing code and modification to avoid or fix a stack overflow. Not all methods apply to every project. Therefore, choose the method that fits your individual needs from the list.

- Avoid using CASE statements due to compiler limitations.
- Avoid complex IF statements where possible.
- Limit the number and size of passed variables.
- Limit the number of local variables.

B.2.1 Avoid CASE Statements

CASE statements are not the most efficient coding practice. This leads to increased use of stack space to store the interim results of each possible SWITCH state. Sample code is shown in Code 1, which is replaced with the code shown in Code 2.

Code 1. Sample Code that Leads to Increased Use of Stack Space

```
switch ( bSwitchVariable )

  {
     case (0x01):    // do something
        break;
     case (0x02):    // do something else
        break;
     default:        // do third thing
        break;
  }
```

Code 2. Sample Code to Avoid Increased Use of Stack Space

```
  If ( bSwitchVariable == 0x01 )
  {
     // do something
  }
  else if (bSwitchVariable == 0x02 )
  {
     // do something else
  }
  else
  {
     // do third thing
  }
```

B.2.2 Avoid Complex IF Statements

Another cause of additional bytes on the stack is complex IF statements. Similar to the problem with CASE statements described in the previous section, an IF statement with several decision points results in a byte being pushed to the stack for each comparison. Wherever possible, replace this type of statement with the simplest form possible. A sample code is shown in Code 3, which is replaced with the code in Code 4.

Code 3. Sample Code for a Complex IF Statement

```
if (!(( bVariable == 0x00 )
   ||  ( bVariable == 0x10 )
   ||  ( bVariable == 0x20 )
   ||  ( bVariable == 0x30 )
   ||  ( bVariable == 0x40 )
   ||  ( bVariable == 0x50 )
   ||  ( bVariable == 0x60 )
   ||  ( bVariable == 0x70 )))
{
   // do something

}
```

Code 4. Sample Code for a Simple IF Statement

```
if (!(( bVariable & 0x8F ) == 0x00))
{
   // do something
}
```

B.2.3 Limit Number and Size of Passed Variables

Each passed variable is pushed to the stack for storage during a function call in C. Therefore, to reduce stack usage, limit the number and size of passed variables. One solution is a global variable, which is stored once in a fixed RAM location and is accessible by every function, adding nothing to the stack. You can pass a pointer to a local variable rather than passing the variable or declaring a global, However, in a PSoC1 device a pointer is two bytes which doubles the effect on the stack. A pointer should be used only for passing the first address of a buffer or an array so that only the 2-byte address, rather than the entire contents of the array, is placed on the stack.

B.2.4 Limit Number of Local Variables

Each local variable used by a function is allocated a byte on the stack when it enters a function. As a result, make sure to limit the number of local variables used by a function. An example of this code is shown in Code 5, which is replaced with the code shown in Code 6.

Code 5. Sample Code with Many Local Variables

```
void SampleFunction ( * abMessageBuffer, bMessageLength )
{
   // local variables
   BYTE bMessageAddress;
   BYTE bMessageType;
   BMessageAddr = abMessageBuffer[0];

   if ( bMessageAddr == 0x01 )
   // do something

   bMessageType = abMessageBuffer[1];
   if ( bMessageType == 0x01 )
   // do something else

   return;

}
```

Code 6. Sample Code with Limited Local Variables

```
void SampleFunction ( * abMessageBuffer, bMessageLength )
{
   if ( abMessageBuffer[0] == 0x01 )
   // do something

   if ( abMessageBuffer[1] == 0x01 )
   // do something else

   return;
}
```

C Appendix C: Legacy Hardware

This section archives some of the legacy debugging hardware that Cypress no longer actively supports. Cypress recommends upgrading legacy hardware to the latest version(s) of hardware described in this application note.

C.1 ICE-4000

The ICE-4000, which is the precursor to the ICE-Cube, has not been supported since PSoC Designer 4.4 and PSoC Programmer 2.3.

The ICE-4000 connects to a PC via a parallel port and was primarily used to debug legacy CY8C25xxx and CY26xxx parts (not recommended for new designs).

Figure 30. ICE-4000

C.2 CY3240 I2USB Bridge Kit

The CY3240 kit supports I2C to USB bridging, but does not provide any programming functionality. The kit is still supported by Cypress software tools, but new users are encouraged to buy the CY8CKIT-002 MiniProg3 kit instead. The MiniProg3 supports I2C to USB bridging and is a superset of the CY3240 capabilities.

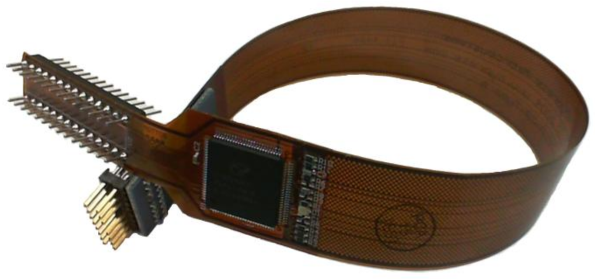
Figure 31. CY3240 I2USB

C.3 Flex Pods

Flex pods are another common piece of legacy debug hardware. They have a standard ICE connector on one end of a flex cable, an OCD part mounted on the flex cable itself, and a PDIP connector at the other end of the cable (used for prototyping).

This hardware is no longer supported by the ICE and PSoC Designer. Flex pods should be upgraded to the latest version of CY3250 pods or CY3210-EvalPods.

Figure 32. Legacy Flex Pod



Document History

Document Title: AN73212 - Debugging with PSoC® 1

Document Number: 001-73212

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	3450775	DRSW	12/15/2011	New application note.
*A	3807180	DRSW	11/09/2012	Updated in new template.
*B	4753600	ASRI	05/27/2015	Added reference for CY3215A-DK kit as CY3215-DK is obsolete. Removed the reference for CY3240 I2USB Bridge Kit because it is obsolete at all the instances. Added it to Legacy Hardware. Added PSoC Resources. Updated Debugging Hardware and Setup. Updated Debugging Pods. Updated Table 1. Added Table 2, Table 3 and Table 4. Updated ICE + OCD Device Mounted on Board. Updated Debugging Environment – PSoC Designer IDE. Updated Debugging Controls. Updated 8-bit Thread Options. Updated Map File (.mp). Updated the document as per new template.
*C	5701681	AESATMP9	04/19/2017	Updated logo and copyright.

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

ARM® Cortex® Microcontrollers	cypress.com/arm
Automotive	cypress.com/automotive
Clocks & Buffers	cypress.com/clocks
Interface	cypress.com/interface
Internet of Things	cypress.com/iot
Memory	cypress.com/memory
Microcontrollers	cypress.com/mcu
PSoC	cypress.com/psoc
Power Management ICs	cypress.com/pmic
Touch Sensing	cypress.com/touch
USB Controllers	cypress.com/usb
Wireless Connectivity	cypress.com/wireless

PSoC® Solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP | PSoC 6

Cypress Developer Community

Forums | WICED IOT Forums | Projects | Videos | Blogs | Training | Components

Technical Support

cypress.com/support

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709