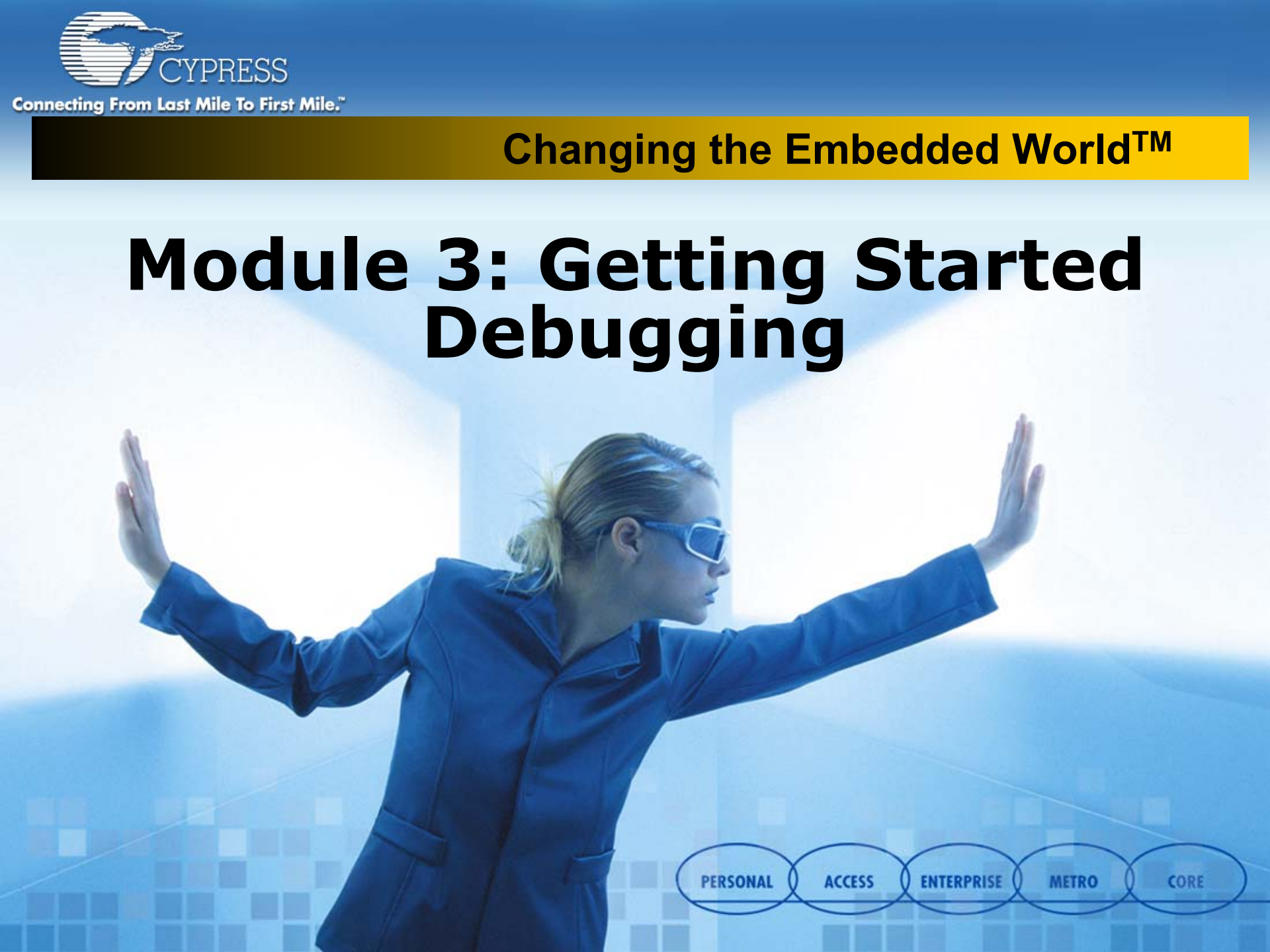
CYPRESS
Connecting From Last Mile To First Mile.™

Changing the Embedded World™

Module 3: Getting Started Debugging



Module Objectives:

Section 1: Introduce Debugging Techniques
Section 2: PSoC In-Circuit Emulator (ICE)
Section 3: Hands on Debugging a System with PSoC MCU

- **Breakpoints**
- **Watch Variables**
- **Dynamic Event Points**

Let's Run our Program

In Circuit Emulators (ICE) required for "real application" testing

Most programs don't work the first time…

You need to test the program in circuit

Debugging allows the developer to monitor the application code line by line as it is running on the circuit.

Most common types of errors

Off by one –

Memory corruption

Timing interrupts – Where did the code go?

Stray pointers

Hardware design errors

Peripheral errors

PSoC Debugging Features Overview

- Both C and Assembly level debugging
- Real time (24MHz) In Circuit Emulation for all part types
- Trace Buffer - 128K Deep
- Breakpoints
- Watch Variables
- Viewing Registers, RAM, Flash,CPU registers
- Dynamic Event Points – more than a Breakpoint
  - Break
  - Trace On
  - Trace Off
  - External trigger

Debugging Features – Code Debugging

Assembly and C level debugging

Assembly –

- Singular step program execution capability
- Displays Instruction in trace buffer
- Full breakpoint capability in assembly

Source level C -

- Single step instruction
- Step over a procedure
- Step out of a procedure
- C – level breakpoint capability

Debugging Features - Trace Buffer

Trace Buffer 128K bytes – (overwrites)

Default status is Trace buffer “ON”

Selectable fields – (Debug→Trace mode)

- PC Program Counter
- Time Stamps
- SP – Stack Pointer
- A – Accumulator
- External pins on ICE

On/Off capability with Dynamic Event Points

Debugging Features - Breakpoints

Halts program execution, provides the user with real time control of program execution.

Selectable via application editor or debugger

Selectable by clicking in left margin of code

Breakpoint manager window

Debugging Features - Watch Variables

Provides the user the ability to View Global variables

Watch Variables may be modified

Watch Variables automatically recalled

Debugging Features - Viewable Items

Access to I/O Registers, RAM, Flash,CPU registers

- CPU Registers – PC,SP,X, A, Flags

Bank 0 and Bank 1 viewable and modifiable

- 512 configuration registers may be changed
- Debugging capability to reconfigure your device while debugging
- Bank 0 Port _2_Data output to LCD's

RAM - Modifiable and viewable when program halts

Flash – Viewable Not modifiable

Debugging Features - Dynamic Event Points

Advanced emulation capabilities –

- Features comparable to $1,000+ emulators
  Provides limitless conditional testing
- Complete conditional TRACE capture
- External triggering
- Provides the capability to sequence complex test scenarios

Similar to a hardware logic analyzer

Why Use Dynamic Event Points?

Problem – How to monitor a complex sequence of events?

- **A standard emulator provides the capability to look at one condition and perform one event a Break.**
- **Dynamic Event Points – Can help you answer "What happens the 20th time through the loop when the Variable is equal to 99h?"**
- **We will perform this level of testing in the hands on section**

Dynamic Event Points – Debugging a sequence of events, instruction by instruction

Each event is a mini if/then – if a condition occurs, then take one or more actions + go to the next conditions

Dynamic Event Points

Typical Dynamic Event Point Usages

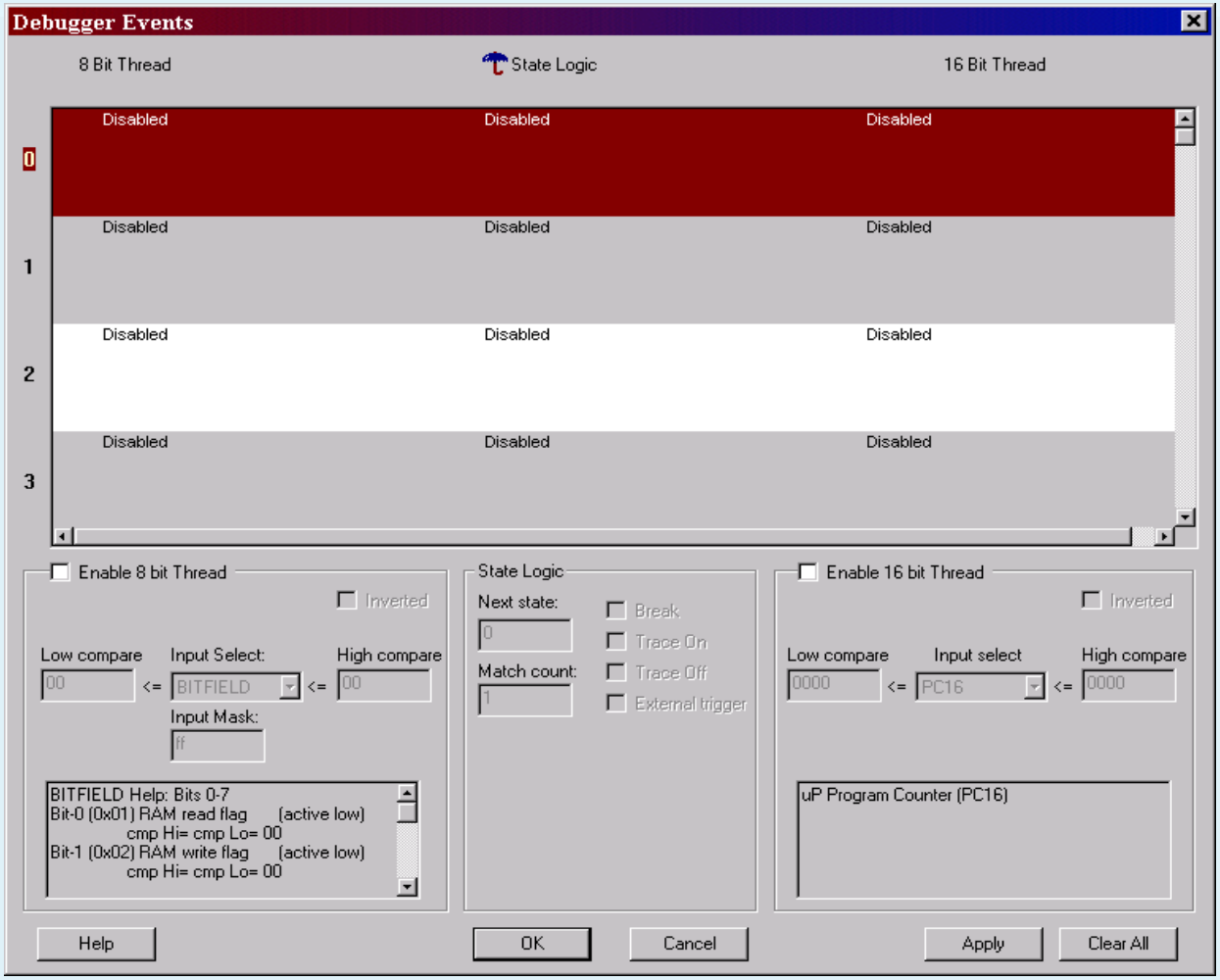
- Find a stack overflow - SP
- Trace a specific range of code - PC
- Find when a memory location is corrupted
- Find when an IO address is written
- Drive an external signal in interrupt
- Measure interrupt latency

Dynamic Event Points

Typical Dynamic Event Point Usages (cont.)

- Break the Nth time a line of code is executed
- Detect jump or call out of program image
- Look for IOX access with specific address
- Break on carry flag status
- Break on signals from customer target board
- Wait for certain number of instructions
- Count sleep periods

Dynamic Event Points: Main window



Conditions – IF …

The test case initialization includes the capability to specify many different inputs and then add conditional logic to them!

The IF – possible inputs

- Accumulator, Stack Pointer, X, PC
- External logic pins
- Data address (Memory or I/O)
- Data value (Memory or I/O)
- Bit Field – Global Int, Extended I/O, R/W, Carry, Zero Flags
- Count

Conditions: Greater Than / Less Than / Equal To

Actions - … THEN

Possible Actions:

Test sequence (event)complete control:

- **ALWAYS – go to next state**
- **Skip a state**
- **Sequence on the same event**

OPTIONS for each event:

- **Turn trace off**
- **Turn trace on**
- **Blip external trigger pin**
- **Break**

Dynamic Event Points: Technical Nuts and Bolts

Multiple threads – 8/16 bit

16 bit threads

- All CPU registers, including PC
- Can monitor address/data at same time
- No masking of bit fields

8 bit threads

- Contains bit field – IOX bit, global int, ram/mem read/write
- Can mask off unwanted bits
- No PC support

Threads can be logically combined

Test suites can be ordered with complete flexibility

Execute Project Within Debugger

The last section of this Module will be hands on. We will use the debugger to find the system bug and demonstrate the ICE features

Steps:

Switch to the Debugger – What's Different?

Connect to the ICE

Download the project

Let's work!

Debugging – Download actions

⇒ Download the GettingStartedproject .rom file to the Pod by clicking the Download to Emulator icon .

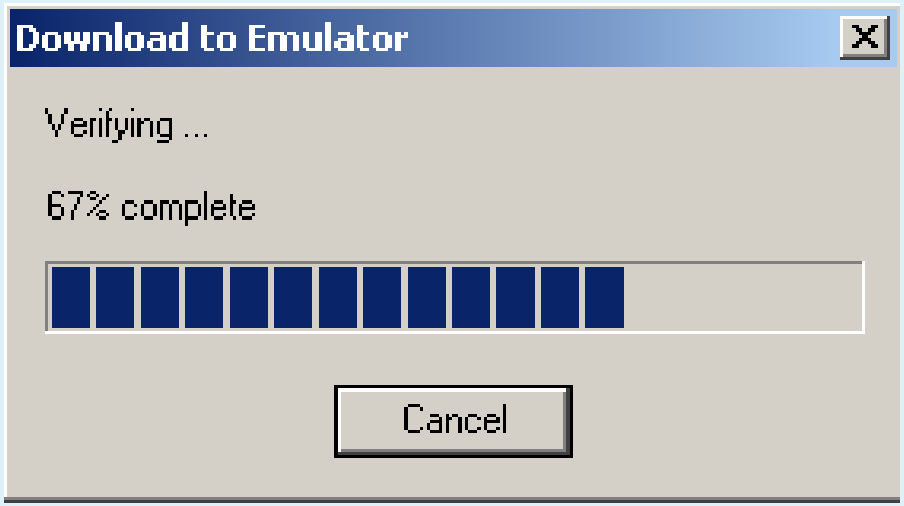
The system automatically downloads your project .rom file located in the …\output folder of your project directory. A progress indicator will report download status.

Upon successful connection, you will receive notification and a green light displaying a status of Connected will display in the lower-right corner of the subsystem.

Emulator Download

Speed has been greatly improved for

- **Emulator download**
- **Device programming**



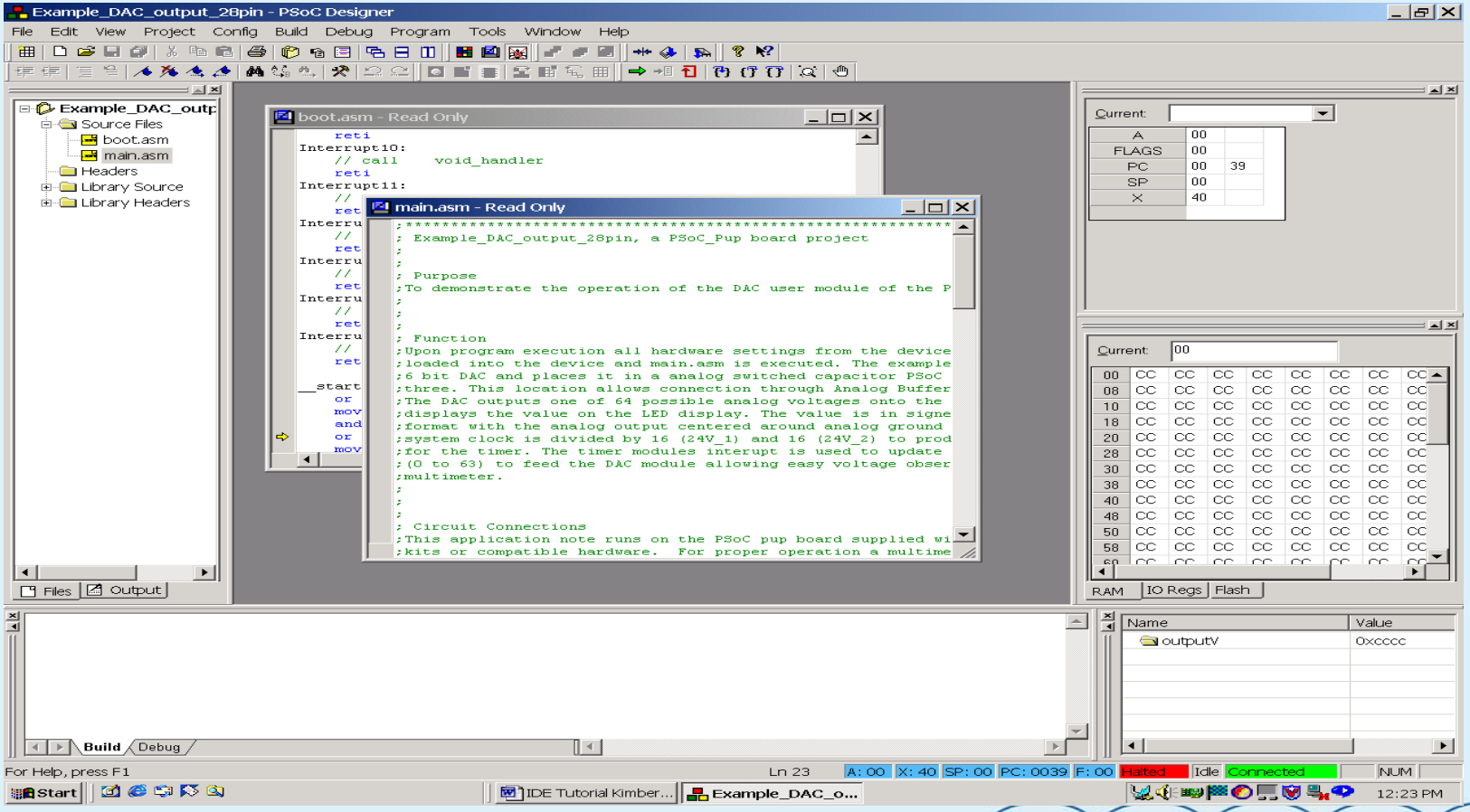
Explore the Debugging Window:

⇒The debugging window has many useful features: Register Memory space, Watch Variable list, output files, source files, and CPU registers (see the device Data Sheet, section 2.0 for details).

In the status bar (at the bottom of the screen)you will find ICE connection indication, debugger target state information, and (in blue) Accumulator, X register, Stack Pointer, Program Counter, and Flag register values, as well as the line number of the current code.

Execute Project Within Debugger

⇒Explore the Debugging Window:



Debugging Overview

Following is a summary of our debug strategy:

First, Compile and Link the project to ensure no run time errors.

Second, Execute the program (RUN) with the PSoC pup connected.

Verify output (on LED's) to expected output.

System operation should be the full range of 8 bit DAC output at a ¼ second update rate showing on the LED's. Starting at the high end and proceeding down to the digital input of zero.

Debugging Features – Code Debugging

Process:

Set breakpoints on both routines to see the decrementing of the OutputV value as well as the transfer to the Accumulator.

The Watch Variable from the code that we will view/monitor is the "OutputV" value. This value should cycle the entire 8 bit DAC range and then reset again.

We will see the output to Port 2 register.

Lastly, We will introduce Dynamic Event Points

Debugging Features - Breakpoints

Breakpoints

This feature of PSoC Designer allows you to stop program execution at predetermined address locations. When a break is encountered, the program is halted at the address of the break, without executing the address's code. Once halted, the program can be restarted using any of the available menu/icon options.

For our example we will set two breakpoints:

Open the *main.asm* file by highlighting it in the source tree. (If the source tree is not showing, access View >> Project.)

Scroll down in the file to the statement: `M8C_EnableGInt`.

⇒ Go to the left margin next to this statement and left click. A red dot will signify that you have just set a breakpoint.

Debugging Features - Breakpoints

Breakpoints

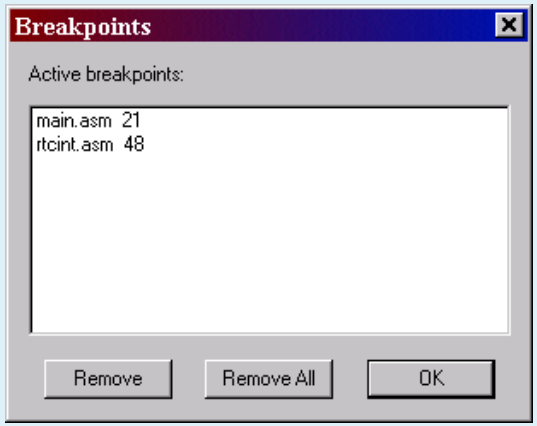
second breakpoint we will set in the RCTINT.*asm* file.

⇒ Scroll down to the statement "`dec [outputV]`"

⇒ Click on the left margin next to this statement. A red dot will signify that you have set a breakpoint.

Breakpoint Manager

To view all the breakpoints you have set, access Debug >> Breakpoints.



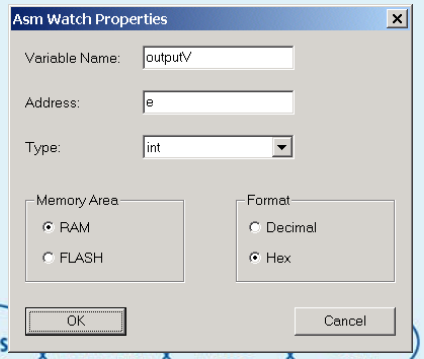
⇒Press OK to close the Breakpoint Dialog Box.

Debugging Features - Watch Variables

The ICE provides the ability to select variables of interest
(from the users program) that can be monitored real time.

We will set a watch on the variable “OutputV.” The address for this variable is 0Eh. This can be found in the Trace window or in the *output.mp* file.

⇒ To set OutputV as a Watch Variable, access Debug >> Watch Variables and fill in the following details:



Trace Window

3 Display Options

Trace Display Can Be:

- Saved to a file
- Viewed, saved and printed as HTML report

CYPRESS MICROSYSTEMS — PSoC Designer Trace Report

C:\Program Files\Cypress MicroSystems\PSoC Designer\Examples\Example_Dynamic_PWM\trace
Wed Apr 17 17:22:01 2002

Trace Report

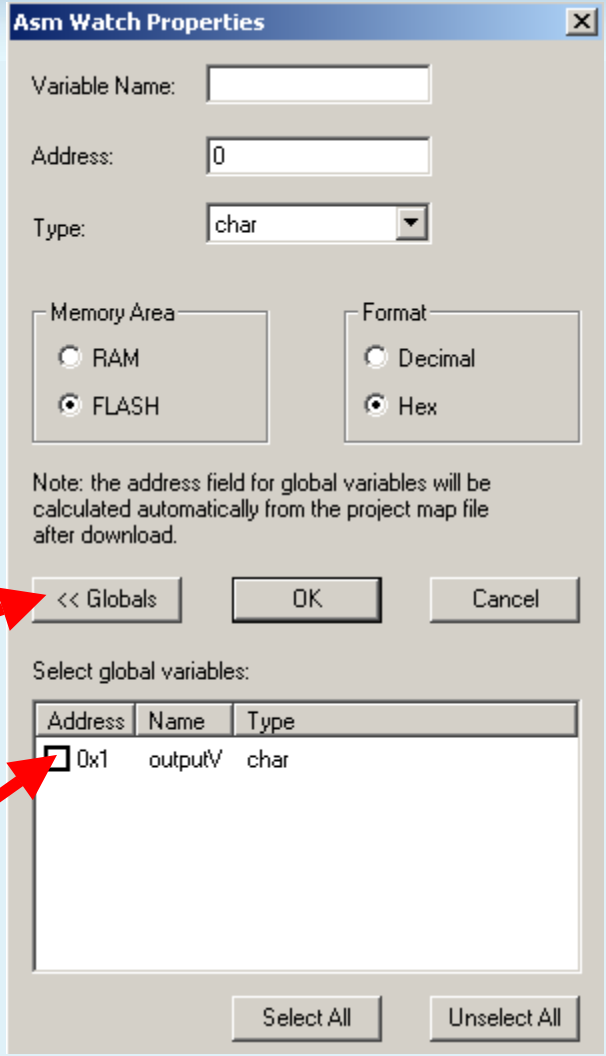
Seq	-- PC --	-- INSTRUCTION --	-- DATA --	-- A --	-- X --	-- SP --	-- FLAGS --	-- EXTERNAL --
0	01C8	OR REG[3Bh], 01h	FF	01	00	06	00	00000000
1	0106	CALL _PWM16_4_Start	08	01	00	06	00	00000000
2	0269	RET	FF	01	00	04	00	00000000
3	0266	OR REG[33h], 01h	FF	01	00	06	00	00000000
4	0104	CALL PWM16_3_Start	06	01	00	06	00	00000000
5	0307	RET	FF	01	00	04	00	00000000
6	0304	OR REG[2Bh], 01h	FF	01	00	06	00	00000000
7	0102	CALL _PWM16_2_Start	04	01	00	06	00	00000000
8	03A5	RET	FF	01	00	04	00	00000000
9	03A2	OR REG[23h], 01h	FF	01	00	06	00	00000000

-- PC --	-- INSTRUCTION --	-- DATA --	-- A --	-- X --	-- SP --	--FLAGS--	EXTERNAL
012A	OR REG[23h], 01h	FF	07	11	06	00	00000000
010D	CALL _PWM16_5_Start	0F	07	11	06	00	00000000
06A9	RET	FF	07	11	04	00	00000000
06A8	POP A	07	07	11	06	00	00000000
06A7	POP X				07	00	00000000
06A4	OR M[01h], 02h				08	00	00000000
06A2	AND F, EFh				08	00	00000000
069F	OR REG[E1h], FFh				08	10	00000000
069C	AND REG[E1h], 00h	00	C2	C2	08	12	00000000
069A	OR F, 10h	00	C2	C2	08	12	00000000

Save
Save As...
View Trace Report

Watch Window

- Global Variable addresses are automatically entered/updated each time the project is downloaded to the ICE
- Global Variables can be selected from the << Globals selection list
  - Name, Address, Type and Memory Area are automatically set
- Check box to select Watch Variable



ICE Status Bar Read Left to Right

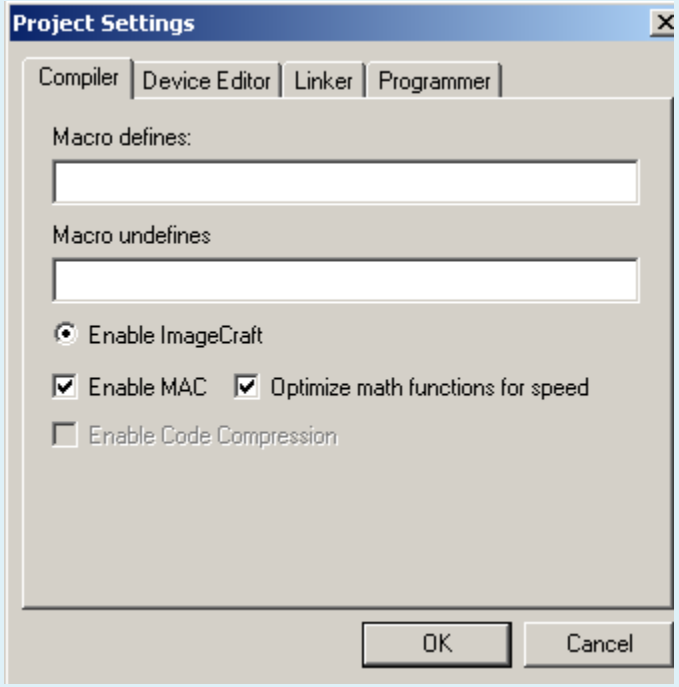
- Accumulator
- Index
- Stack Pointer
- Program Counter
- Flag
- Actual CPU frequency
- Program execution status
- ICE communication status
- ICE connection status
- Carry and Zero flags

C Compiler Project Settings

Macro Defines

Enable Multiply/Accumulate (MAC)

Optimize Math Functions for Speed



C Interrupts

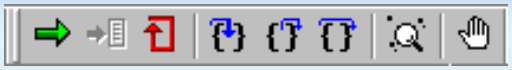
C Interrupts are Supported

#pragma interrupt_handler <name> *

- `reti` is used instead of `ret` to return from the function
- Virtual registers used by the function are automatically saved and restored
- If another function is called from the interrupt handler all virtual registers are saved and restored

Additional Information Available in the C Language Compiler User Guide

Debugging Features - Execute the Program

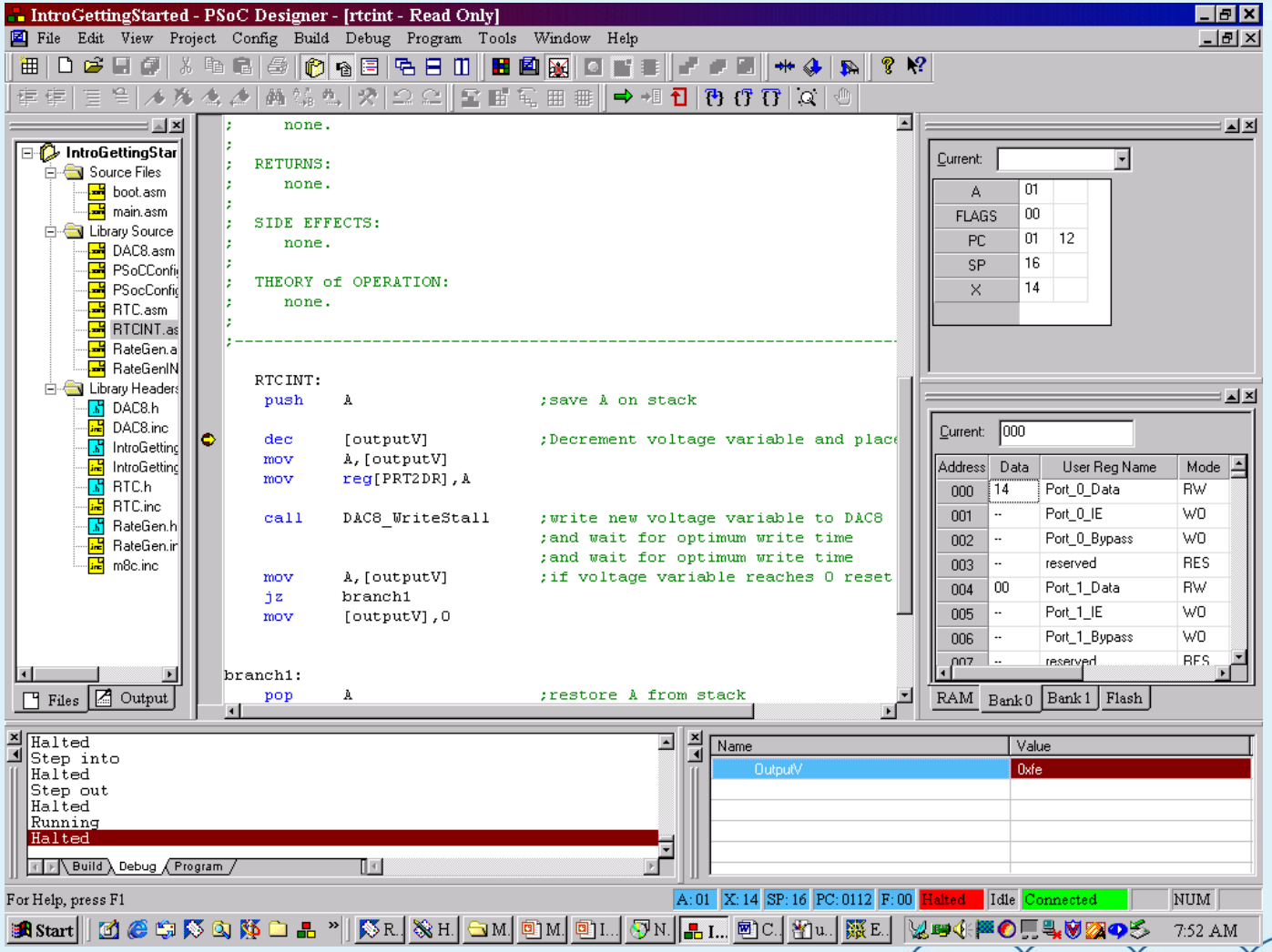


We are now ready to start execution of the program. Our example should stop on the first breakpoint in *main.asm*. A yellow arrow will point to the `M8C_EnableGInt` line of code when this happens.

Use the "Step Into…" functions to execute the next several instructions. Watch what happens to outputV, the Accummulator and the LEDs on the PuP at each step.

Click the Green Run icon to execute the program again. The program will stop on the second breakpoint in RTC*INT.asm*, "`dec [outputV]`"

Debugging screen: Viewable Items



Viewable Items

Access to I/O Registers, RAM, Flash,CPU registers

- CPU Registers – PC,SP,X, A, Flags

Bank 0 and Bank 1 viewable and modifiable

At this point in our tutorial we can view the I/O registers. The results of the "OutputV" variable will be output onto Port 2 data line, which is located at I/O Address Register 008.

In the right frame of the Debugger subsystem, click the Flash (Bank0) tab and go to memory location 008. This is the Port 2 data line, which will be output to the Pod LED display. At this point, the LEDs should be lit representing this value.

View RAM Registers

- ⇒ **View the** OutputV **variable in the RAM section of the Debugger. Click the RAM tab and scroll over to the address location 0E. The value is also shown in the Watch Variable window in the lower right**

- **The Ram values may be dynamically changed while in the debugger.**

System Verification:

Is the system working correctly?

NO!

- If we let the program run we can see that the LED's are not sequencing through the 8 bit DAC values.
- Let's use the latest PSoC feature, Dynamic Event Points to find our run-time error!

Test scenario:

Using test code from GettingStarted several test cases have been developed as examples to demonstrate the usage of Dynamic Event Points.

Test Case #1 – Demonstrates using the 8-Bit and 16-Bit test threads in combination, monitoring an address location, breaking on event and turning trace on. (MEM_DA, MEM_DB).

Test Case #2 – Demonstrates monitoring a specific assembly instruction using the Match count parameter.

Test Case #3 – Monitoring a particular line of code 16-Bit (PC) and looking at the Flag registers.

Dynamic Event Points

Test Case #1: Demonstrating Mem_DA, Mem_DB and the Combinatorial Operator.

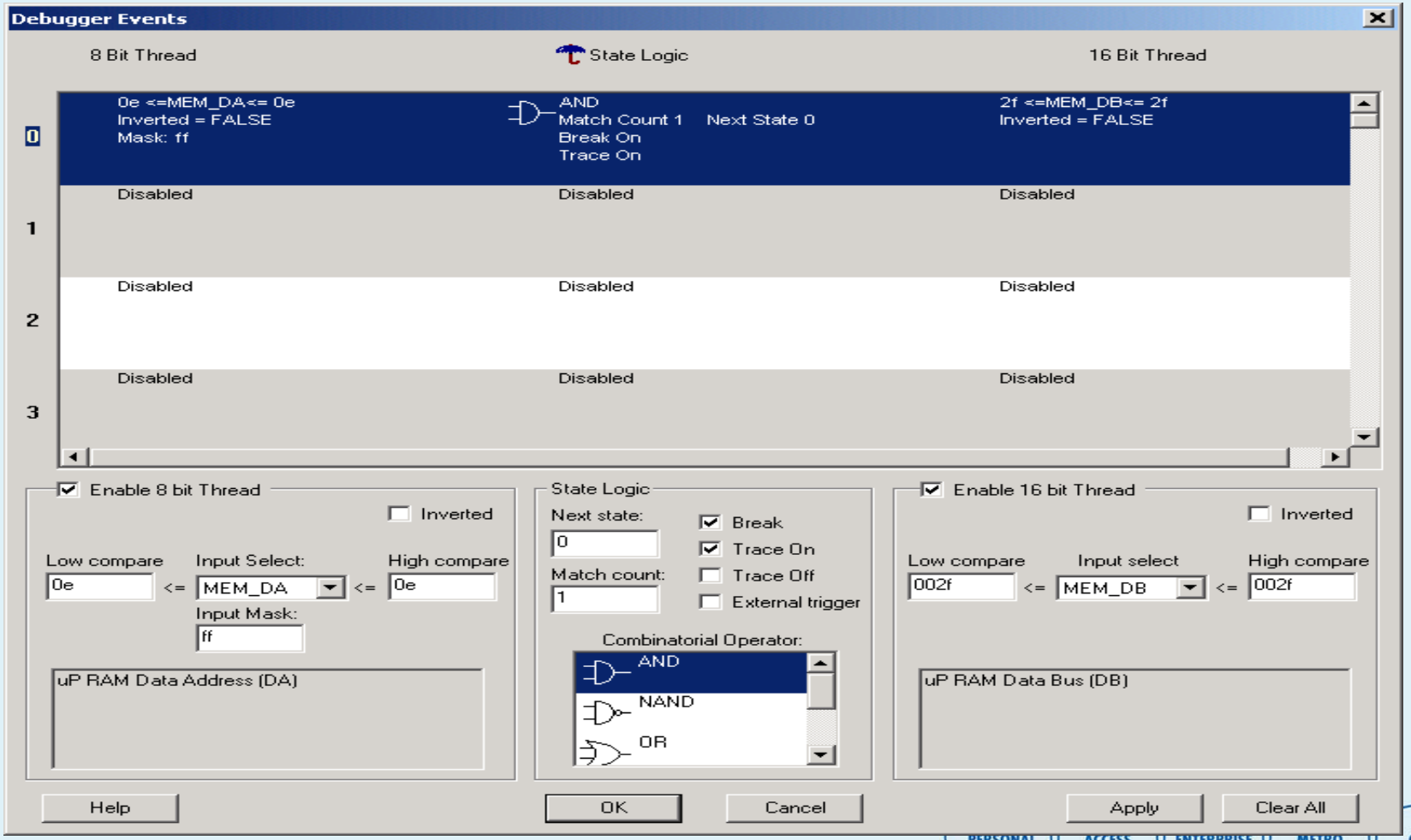
Test: Determine if OutputV ever reaches a mid range of 2f (hex)? If it does: Turn trace on and Break.

The address location for OutputV is 0eH. (This can be found by viewing the .mp file which can be found by selecting the View toolbar →Output.)

Expected test outcome:

A Break TWO LINES BELOW the mov A,outputV line of code, the address location 0e = 2F, and the Accumulator = 2F which would be correct for the program

Dynamic Event Points: Test Case #1



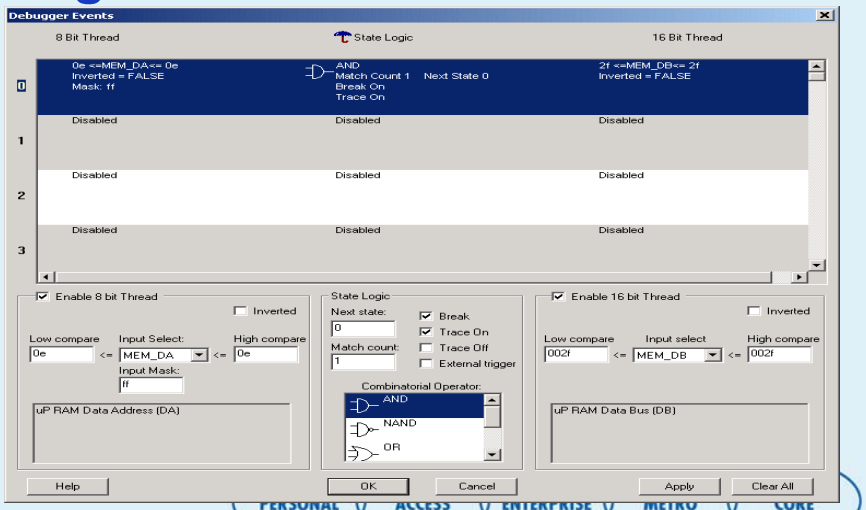
Dynamic Event Points: Test Case #1-results

Test Case #1: We are looking to see if we hit a mid-range of the DAC.

Results: If we were to hit OutputV = 2f the program would break.

It doesn't! Let's try something else.

Is the Decrement instruction working?



Dynamic Event Points

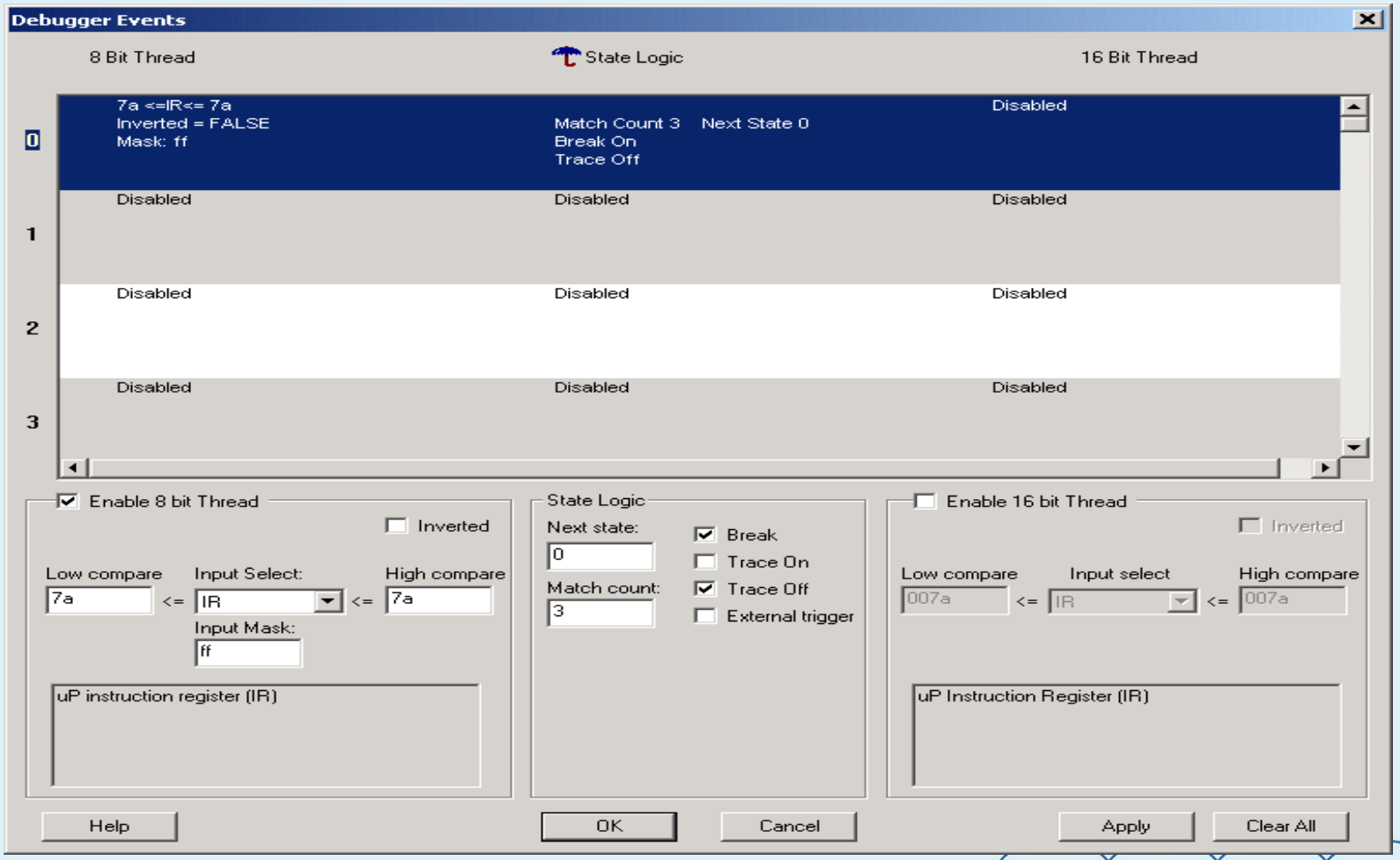
Test Case #2: Demonstrate (IR) Instruction Register, and the Match Count

Test: Let's see if the DEC instruction is working correctly? When the decrement instruction (DEC) is executed 3 times Break Turn trace OFF.

Expected test outcome:

A Break TWO LINES BELOW the DEC command.

Dynamic Event Points: Test Case #2



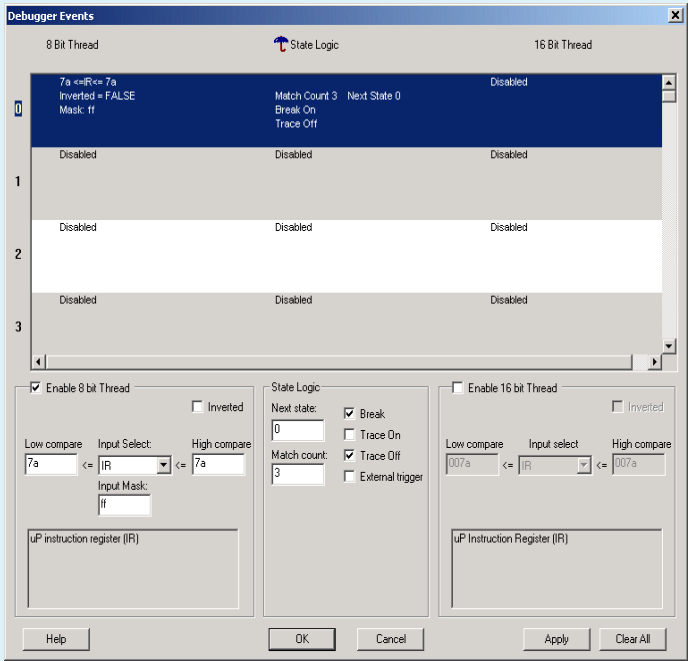
Dynamic Event Points: Test Case #2-results

Test Case #2: We are looking to see if the decrement command is cycling

Results: Do we hit a breakpoint relatively quickly?

Yes! So we know that our cycle should be working

Let's try something else.



Dynamic Event Points

Test Case #3: Let's revisit what this program should do!

We know from the previous tests the following: The mid range never is hit(2F), But - the DEC command seems to be working.

Dynamic Event Points will enable a more complex testing to find our problems: Hint there are two!

Dynamic Event Points

Test Case #3: Let's start with a simple code inspection first.

Why isn't OutputV ever hitting the high end?

Answer: A code review of main.asm and RCTINT.asm shows that we start with 255 and then we DEC first. So our effective range is only 254.

OK that's one bug, Now why isn't it cycling?

Dynamic Event Points: Test Case #3

Let's perform a conditional branch test. (Many regulated environments require this type of testing)

Let's look and see if Branch 1 is ever executed?

Also let's look to see if how often the resetting of OutputV occurs. It should only occur after the completion of the loop 255-0.

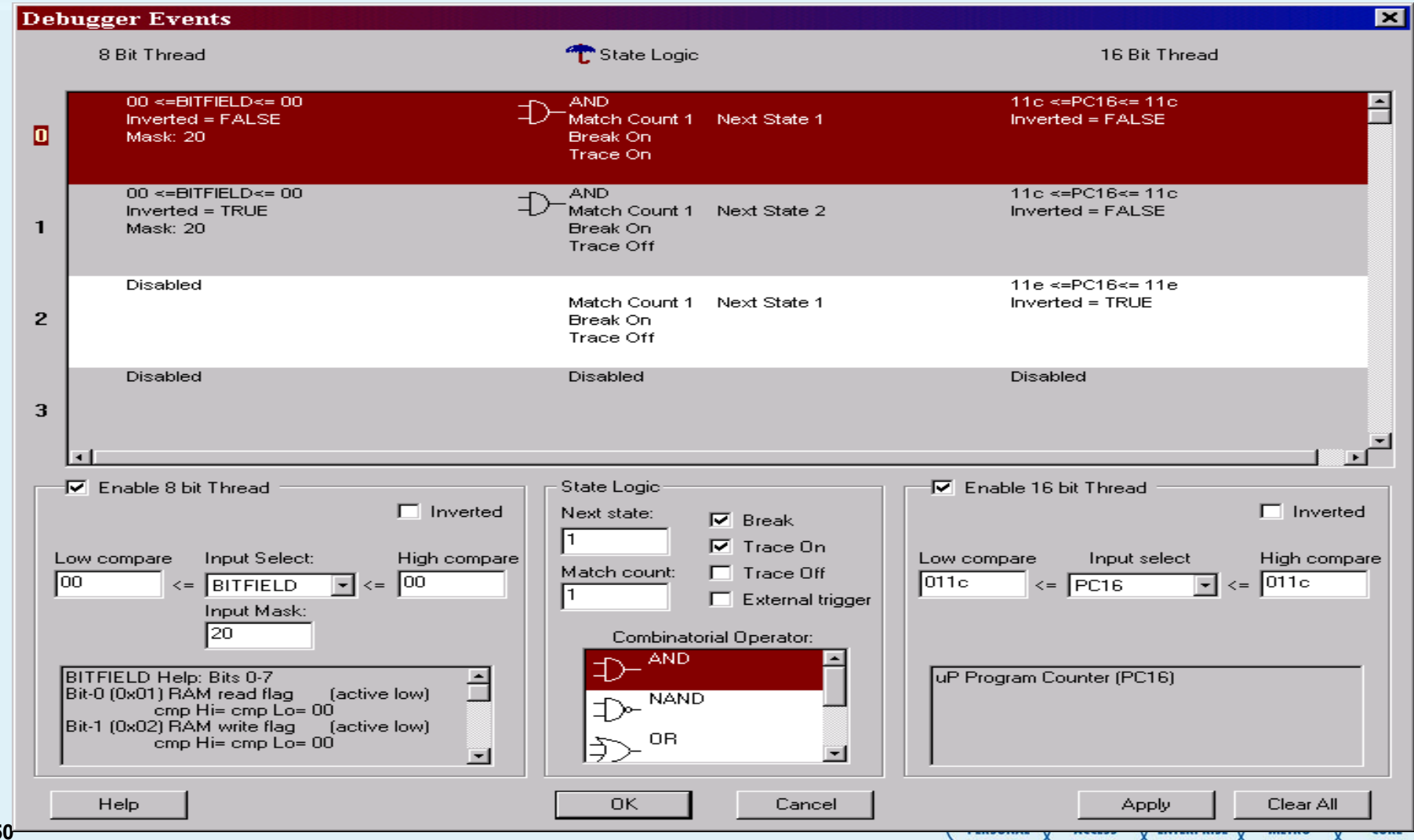
Look at the .lst file (in the output section) for the PC values. Let's investigate the conditions of the jz command as well.

Dynamic Event Points: Test Case #3

Example List file (Your's may be slightly different- do a Control Find for RTCINT)

```
0054) mov  A,[outputV]    ;if voltage variable reaches 0 reset to
      maximum value

   011A: 51 0E    MOV   A,[outputV]

(0055) jz  branch1

   011C: A0 04    JZ    0x0121

(0056) mov  [outputV],0

   011E: 55 0E 00 MOV   [outputV],0

(0059) branch1:

(0060) pop  A                  ;restore A from stack

   0121: 18       POP   A
```

Dynamic Event Points: Test Case #3



Dynamic Event Points: Test Case #3

Expected Results for our tests:

We are looking in the first sequence for a false on the jz command. If it is false then the code will drop to the next command and reset the OutputV value.

Our next event is looking for the inversion of the above. Does the jz ever hit true?

Dynamic Event Points: Test Case #3

Expected Results for our tests:

Let's let it run and see what happens.

We should get lot of breaks if the code is cycling correctly…….

Answers will be given in the class…..

Let's Hit RUN!

CYPRESS

Connecting From Last Mile To First Mile.™