Software Development and Debug

Zynq

Vivado 2017.1 Version

XILINX.

Objectives

> **After completing this module, you will be able to:**

- Describe device drivers architecture
- Distinguish between Level-1 and Level-2 device drivers
- List types of processor timers
- Understand CPU’s private timer API
- Describe GNU Debugger (GDB) functionality
- Describe Xilinx Microprocessor Debugger (XMD) functionality
- Describe Eclipse Target Communications Framework

Outline

- ***Device Drivers Architecture***
- **Timers and API**
- **Debugging Tools**
  - Hardware Tools
  - Software Tools
- **Debug in SDK**
- **Summary**

Device Drivers

- The Xilinx device drivers are designed to meet the following objectives:
  - Provide maximum portability
    - The device drivers are provided as ANSI C source code
  - Support FPGA configurability
    - Supports multiple instances of the device without code duplication for each instance, while at the same time managing unique characteristics on a per-instance basis
  - Support simple and complex use cases
    - A layered device driver architecture provides both
      - Simple device drivers with minimal memory footprints
      - Full-featured device drivers with larger memory footprints
  - Ease of use and maintenance
    - Xilinx uses coding standards and provides well-documented source code for developers

Drivers: Level 0/Level 1

- **The layered architecture provides seamless integration with…**
  - (Level 2) RTOS application layer
  - (Level 1) High-level device drivers that are full-featured and portable across operating systems and processors
  - (Level 0) Low-level drivers for simple use cases

Level 2, RTOS Adaptation
Level 1, High-level Drivers
Level 0, Low-level Drivers

Drivers: Level 0

- **Consists of low-level device drivers**
- **Implemented as macros and functions that are designed to allow a developer to create a small system**
- **Characteristics:**
  - Small memory footprint
  - Little to no error checking is performed
  - Supports primary device features only
  - No support of device configuration parameters
  - Supports multiple instances of a device with base address input to the API
  - Polled I/O only
  - Blocking function calls

Drivers: Level 1

- **Consists of high-level device drivers**
- **Implemented as macros and functions and designed to allow a developer to utilize all of the features of a device**
- **Characteristics:**
  - Abstract API that isolates the API from hardware device changes
  - Supports device configuration parameters
  - Supports multiple instances of a device
  - Polled and interrupt driven I/O
  - Non-blocking function calls to aid complex applications
  - May have a large memory footprint
  - Typically, provides buffer interfaces for data transfers as opposed to byte interfaces

Comparison Example

UARTPS Level 1

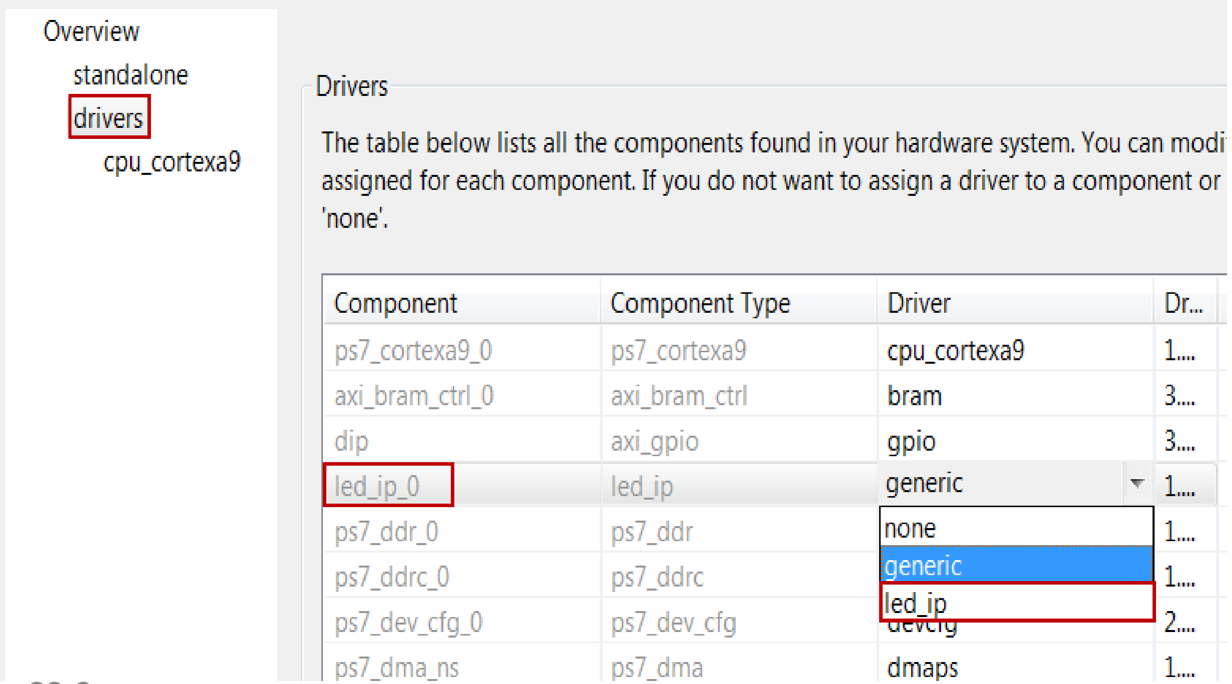
- **XUartPs_CfgInitialize()**
  - Initializes a specific XUartPs instance
- **XUartPs_Send()**
  - Sends the specified buffer using the device
  - polled or interrupt mode
- **XUartPs_Recv()**
  - Receive a specified number of bytes
  - store it into the specified buffer
- **XUartPs_SetBaudRate()**
  - Sets the baud rate

UARTPS Level 0

- **XUartPs_SendByte()**
  - Sends one byte
- **XUartPs_RecvByte()**
  - Receives one byte

Driver Settings

- Select the Drivers panel
- By default, the Driver panel displays which device driver is used for each hardware instance in the design
- Enables selection of custom drivers and versions for each device in the design



Outline

- Device Drivers Architecture
- ***Timers and API***
- Debugging Tools
  - Hardware Tools
  - Software Tools
- Debug in SDK
- Summary

XILINX

Timers: Cortex-A9 Processor

- Timers are an important part of an embedded system
- CPU Private Timer and Watchdog Timer
- Global timer (GTC)
- Two 16-bit triple timer counter (TTC)
- System watchdog timer (SWDT)

Private Timer/Counter (Standalone)

- By default 32-bit count down timer
- xscutimer.h, xscutimer_hw.h header files
- XScuTimer_LookupConfig()
  - Looks up the device configuration based on the unique device ID
- XScuTimer_CfgInitialize()
  - Initialize a specific XTtcPs instance such that the driver is ready to use
- XScuTimer_Start() – Start the timer
- XScuTimer_Stop() – Stop the timer
- XScuTimer_GetPrescaler() – Get the pre-scalar value
- XscuTimer_SetPrescaler() – Set the pre-scalar value between 1 and 16

Private Timer/Counter (Standalone)

- XScuTimer_EnableAutoReload() – Load the counter with the initial value when time out occurs
- XScuTimer_IsExpired() – Check if the timer has reached the final value
- XScuTimer_RestartTimer() – Read the counter value and write it back
- XScuTimer_LoadTimer() – Load the timer with the provided value
- XScuTimer_GetCounterValue() – Get current counter value; useful for determining lapse time
- XScuTimer_EnableInterrupt() – Enable interrupt mechanism
- XScuTimer_GetInterruptStatus() – Get the interrupt status
- XScuTimer_ClearInterruptStatus() – Clear source of interrupt flag

Timers: Triple Timer Counter API (Standalone)

> XTtcPs_LookupConfig() - Looks up the device configuration based on the unique device ID

> XTtcPs_CfgInitialize() - Initialize a specific XTtcPs instance such that the driver is ready to use

> XTtcPs_SetMatchValue() - Set the value of the match registers

> XTtcPs_SetOptions() - Set the options for the TTC device

> XTtcPs_SetPreScalar() - Set the prescalar enable bit

> XTtcPs_GetMatchValue() - Get the value of the match registers

> XTtcPs_GetOptions() - Gets the settings for the options for the TTC device

> XTtcPs_GetPrescaler() - Gets the input clock prescalar

AXI Timer

- XTmrCtr_Initialize() - Initialize a specific timer/counter instance/driver
- XTmrCtr_InterruptHandler() - Interrupt Service Routine (ISR) for the driver
- XTmrCtr_SetHandler() - Sets the timer callback function, which the driver calls when the specified timer times out
- XTmrCtr_GetOptions() - Enables the specified options for the specified timer counter
- XTmrCtr_Start() - Starts the specified timer counter of the device such that it starts running
- XTmrCtr_Stop() - Stops the timer counter by disabling it
- XTmrCtr_GetCaptureValue() - Returns the timer counter value that was captured the last time the external capture input was asserted

AXI Timer

> XTmrCtr_GetOptions() - Get the options for the specified timer counter

> XTmrCtr_GetStats() - Get a copy of the XtmrCtrStats structure, which contains the current statistics for this driver

> XTmrCtr_Getvalue() - Get the current value for the timer counter

> XTmrCtr_Reset() - Reset the specified timer counter of the device

Outline

> Device Drivers Architecture
> Timers and API
> ***Debugging Tools***
>> Hardware Tools
>> Software Tools
> Debug in SDK
> Summary

Debugging

> **Debugging is an integral part of embedded systems development**

> **The debugging process is defined as testing, stabilizing, localizing, and correcting errors**

> **Two methods of debugging:**

  >> Hardware debugging via a logic probe, logic analyzer, in-circuit emulator, or background debugger

  >> Software debugging via a debugging instrument

    – A software debugging instrument is source code that is added to the program for the purpose of debugging

> **Debugging types:**

  >> Functional debugging

  >> Performance debugging

XILINX

Software Debugging Support

- Vivado/SDK supports software debugging through
  - GDB tools
    - Unified graphical interface for debugging and verifying processing systems
  - Xilinx Microprocessor Debugger (XMD)
    - Runs all the hardware debugging tools and communicates with the hardware
    - Shell for hardware communication
    - Tool command language (Tcl) syntax and command interpreter
  - GNU tools communicate with the hardware through XMD
  - Xilinx System Debugger, Eclipse Target Communications Framework (TCF)

Hardware Debugging Support

- **Vivado supports hardware debugging via the following tools**
  - Vivado Logic Analyzer software
    - Soft-core base logic analyzer
    - Operates through a Xilinx download cable
- **Zynq™ AP SoC virtual platform**
  - Functional simulation of physical hardware for the purpose of software development, integration, and test
  - Runs on the desktop
  - Facilitates early software development and test
- **Zynq AP SoC open-source QEMU model**
  - Open-source machine emulator and virtualizer for Linux environment

XMD Debugger

- **The Xilinx Microprocessor Debug (XMD) utility provides a variety of user debug services**
  - Physical connection between your workstation and the software design
  - Connection to an internal BSCAN controller
  - Program download
  - Processor identification and control
  - Low-level debug commands
  - Interface to the GNU debugger
  - General Tcl interface and command interpreter

XMD Functionality

- **XMD engine**
  - Program that facilitates a unified GDB interface
  - Tcl interface and command interpreter
- **XMD supports application debugging on different targets**
- **GDB/TCF can connect to XMD on the same computer or on a remote computer on the Internet**

XMD Commands

- There are many XMD commands
- Popular commands for boot and program control
  - connect – connect to processor
  - dow – download ELF executable file
  - elf_verify – verify ELF file with memory image
  - run – begin program execution from reset
  - con – continue program execution from current program counter
  - stop – stop the target processor
  - exit – close XMD window
- XMD will search for a processor when started and launched from the SDK Run > Debug menu
- connect command will execute automatically
  - connect arm 64

XMD Tcl Inteface

- xhelp: Lists all Tcl commands
- xrmem target addr [num]: Reads num bytes or 1 byte from the memory *addr*
- xwmem target addr value: Writes an 8-bit byte value at the specified memory *addr*
- xrreg target [reg]: Reads all registers or only register number *reg*
- xwreg target reg value: Writes a 32-bit value into register number *reg*
- xdownload target [-data] filename [addr]: Downloads the given ELF or data file (with -data option) onto the memory of the current target
- xcontinue target [addr]: Continues execution from the current PC or from the optional address argument

System Debugger

> **Eclipse Target Communication Framework**
>> Extensible network protocol for communicating with embedded systems

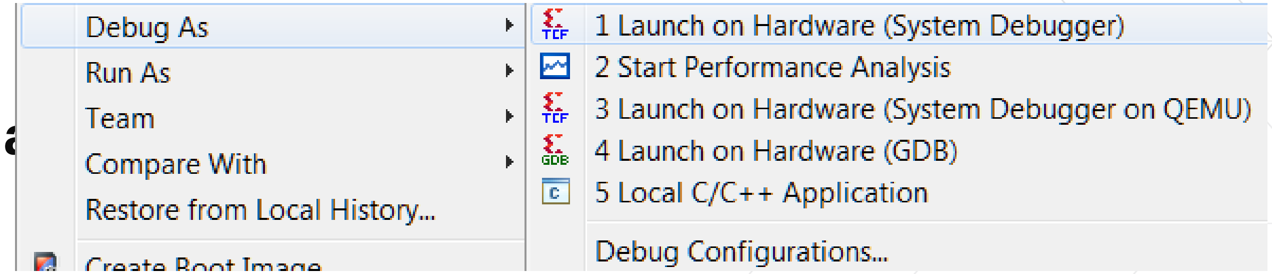
> **Single Configuration per target**
>> (Not per tool like gdb)

> **Homogenous, and heterogeneous, SMP a**

> **Neon Support**

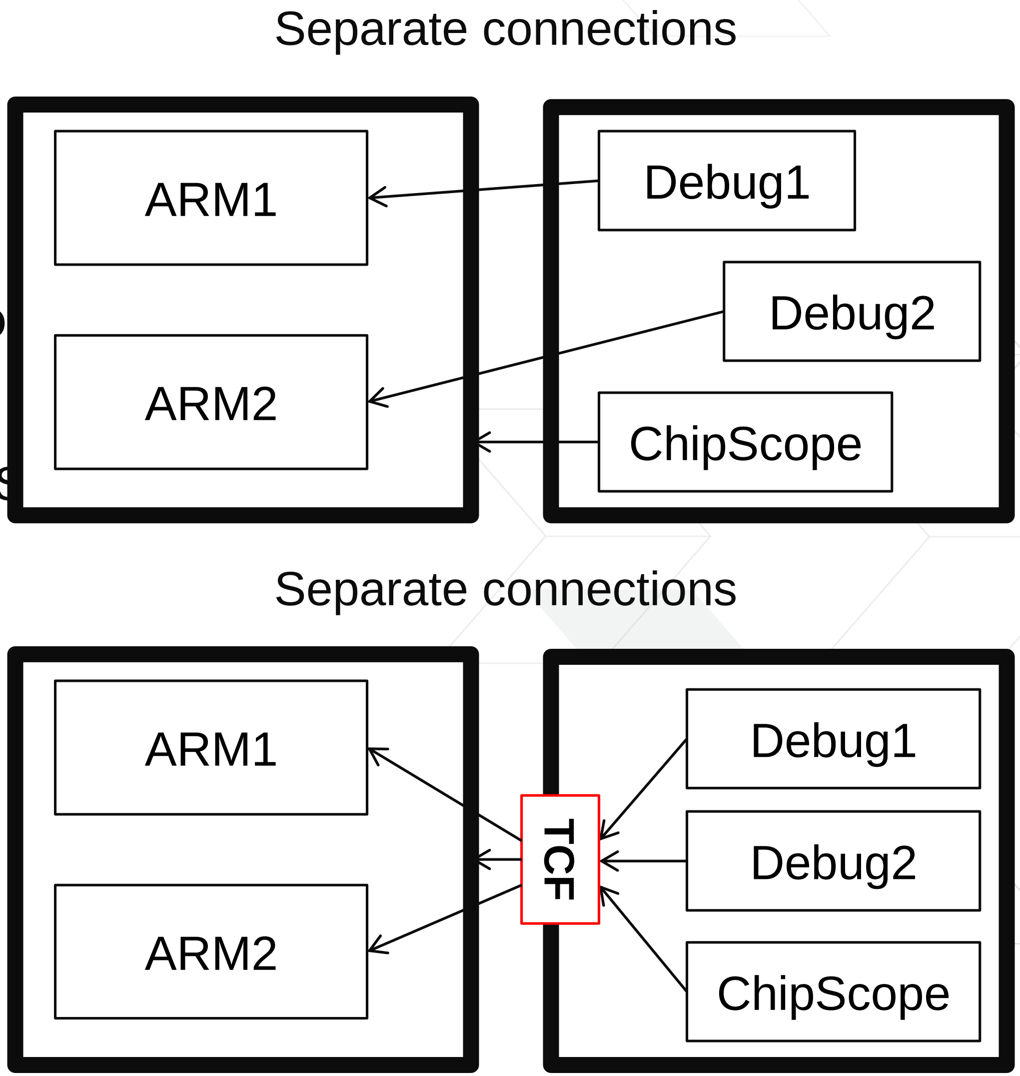
> **True multicore debug through a single JTAG**

> **Faster than GDB/XMD**



Eclipse Target Communication Framework

- Open, extensible network protocol
- Allows services to transparently plug in
- All communication links can share the same proto
- Transport-agnostic channel abstraction
  - (No specific transport layer. E.g. TCP/IP, Serial Line, SS

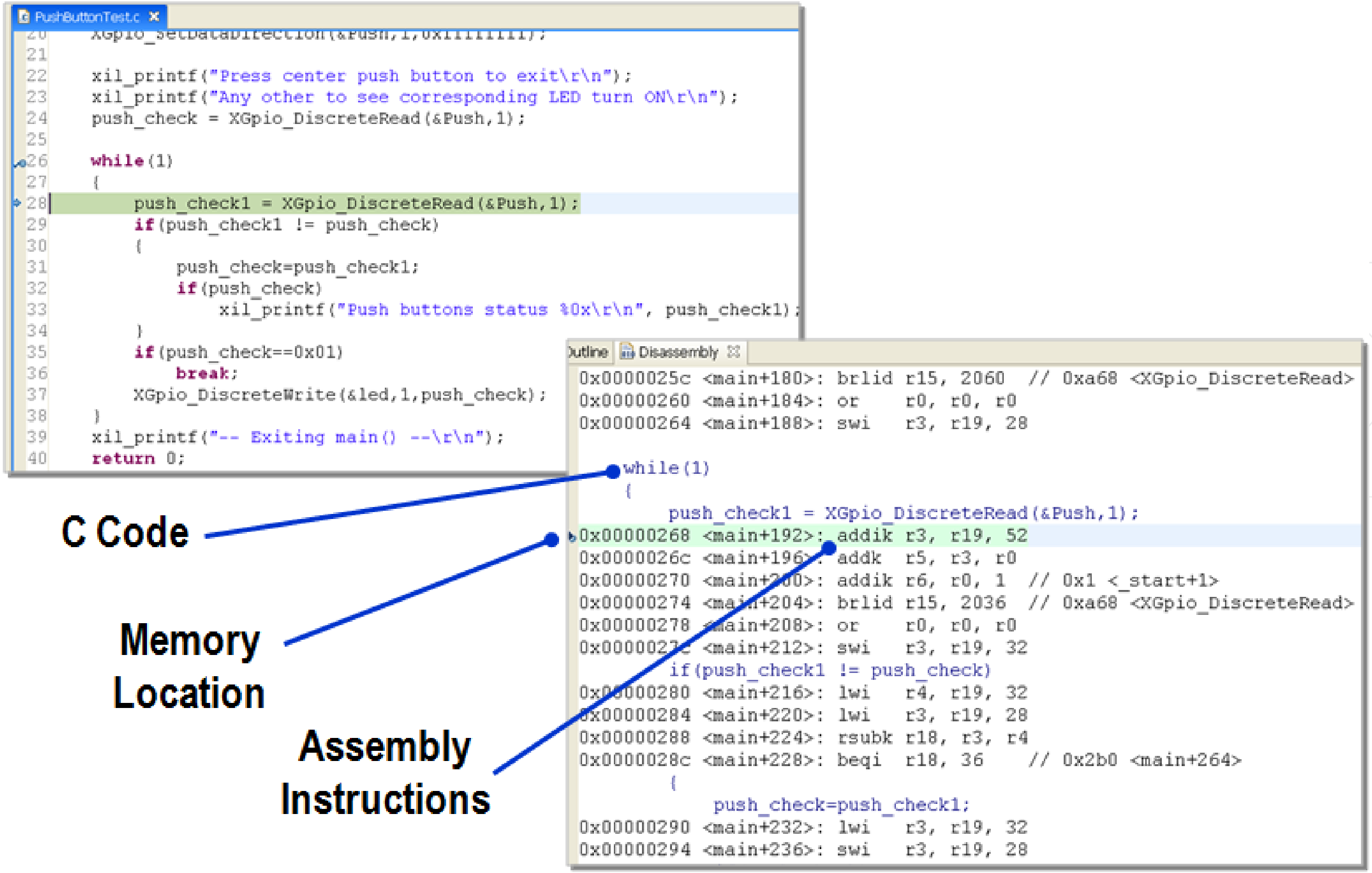


XILINX

Debug Functionality

- Source-level debugger (GDB/System Debugger)
- Start your program
  - Set breakpoints (make your program stop on specified conditions)
  - Examine what has happened, when your program encounters breakpoints
    - Registers
    - Memory
    - Stack
    - Variables
    - Expressions
  - Change things in your program so that you can experiment with correcting the effects of one bug and go on to another
- You can debug programs written in C and C++

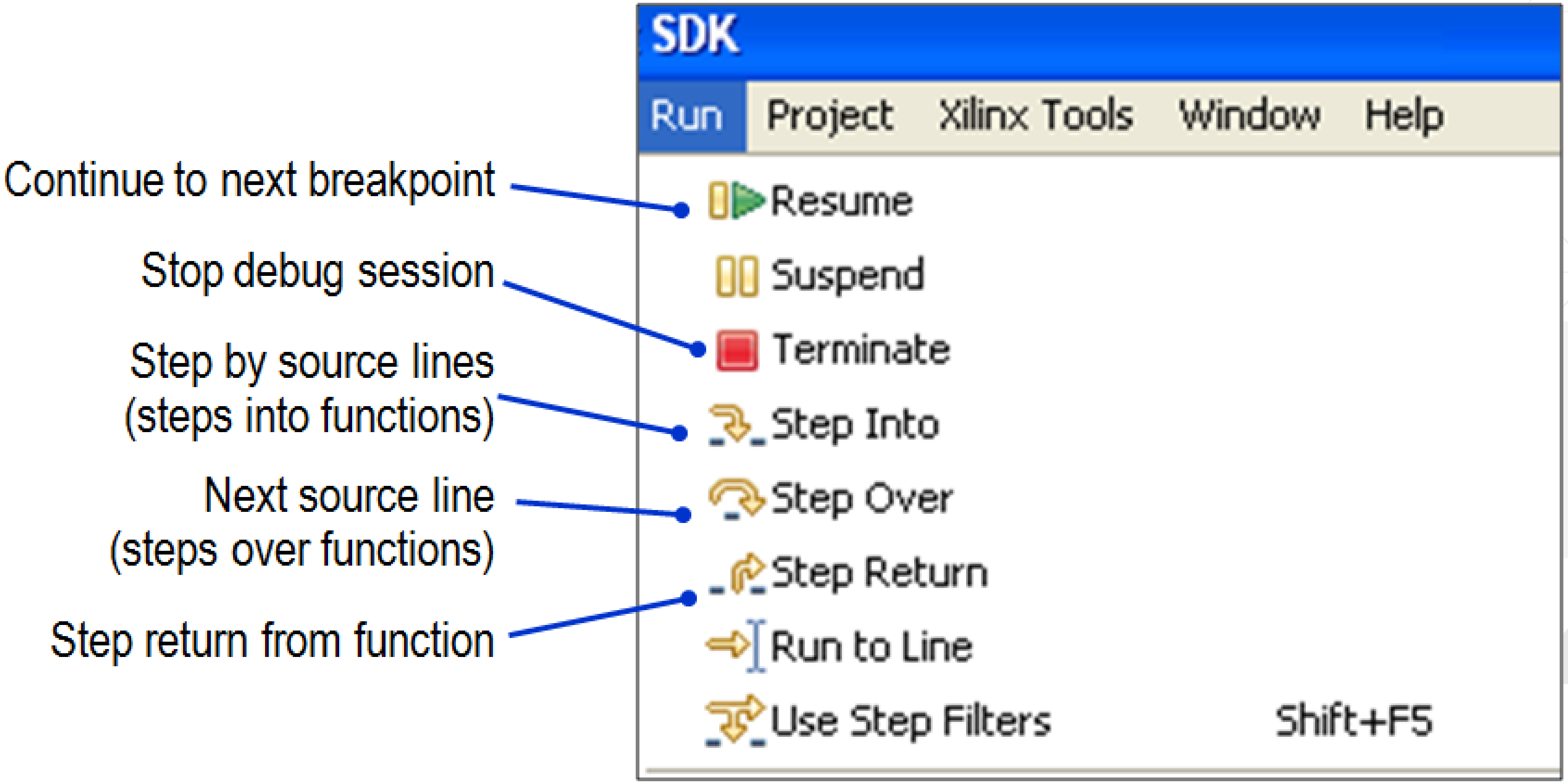
Debug



PushButtonTest.c
xil_printf("Press center push button to exit\r\n");
xil_printf("Any other to see corresponding LED turn ON\r\n");
push_check = XGpio_DiscreteRead(&Push,1);
while(1)
{
push_check1 = XGpio_DiscreteRead(&Push,1);
if(push_check1 != push_check)
{
push_check=push_check1;
if(push_check)
xil_printf("Push buttons status %0x\r\n", push_check1);
}
if(push_check==0x01)
break;
XGpio_DiscreteWrite(&led,1,push_check);
}
xil_printf("-- Exiting main() --\r\n");
return 0;
Disassembly
0x0000025c <main+180>: brlid r15, 2060 // 0xa68 <XGpio_DiscreteRead>
0x00000260 <main+184>: or r0, r0, r0
0x00000264 <main+188>: swi r3, r19, 28
while(1)
{
push_check1 = XGpio_DiscreteRead(&Push,1);
0x00000268 <main+192>: addik r3, r19, 52
0x0000026c <main+196>: addk r5, r3, r0
0x00000270 <main+200>: addik r6, r0, 1 // 0x1 <_start+1>
0x00000274 <main+204>: brlid r15, 2036 // 0xa68 <XGpio_DiscreteRead>
0x00000278 <main+208>: or r0, r0, r0
0x0000027c <main+212>: swi r3, r19, 32
if(push_check1 != push_check)
0x00000280 <main+216>: lwi r4, r19, 32
0x00000284 <main+220>: lwi r3, r19, 28
0x00000288 <main+224>: rsubk r18, r3, r4
0x0000028c <main+228>: beqi r18, 36 // 0x2b0 <main+264>
{
push_check=push_check1;
0x00000290 <main+232>: lwi r3, r19, 32
0x00000294 <main+236>: swi r3, r19, 28
C Code
Memory Location
Assembly Instructions

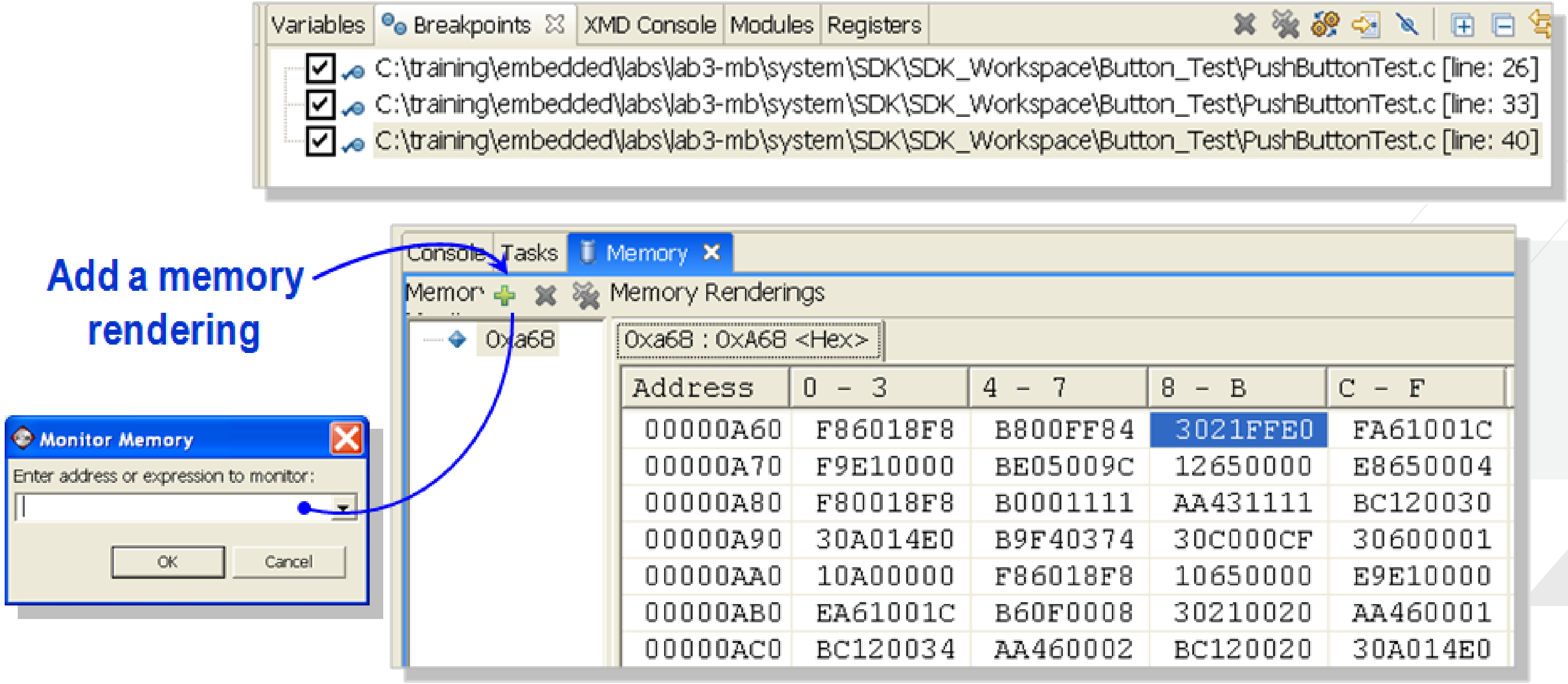
Debug GUI

> **Run-time control**



Debug Functionality

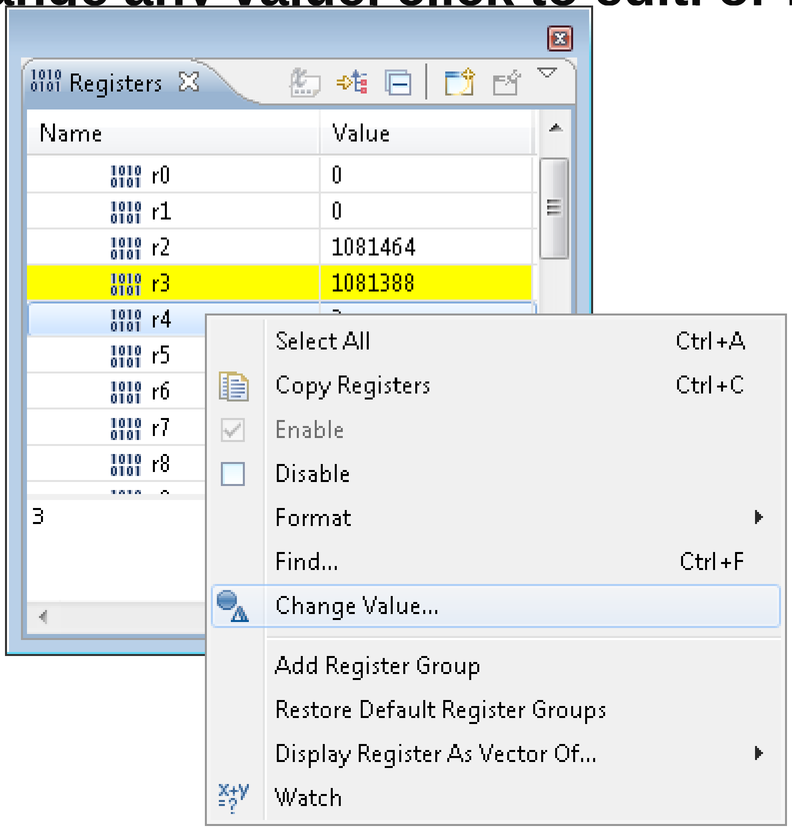
- Breakpoints can be enabled or disabled
- To change any memory value, click a memory field

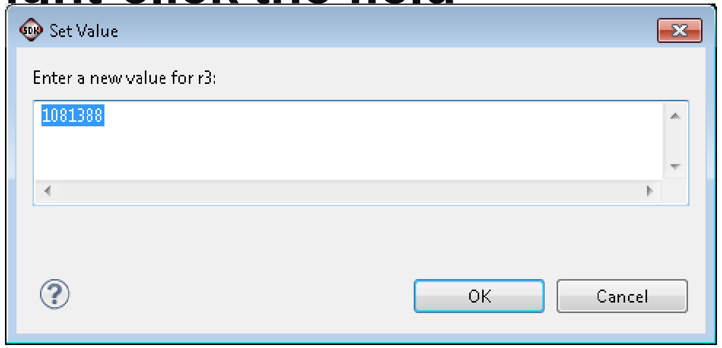


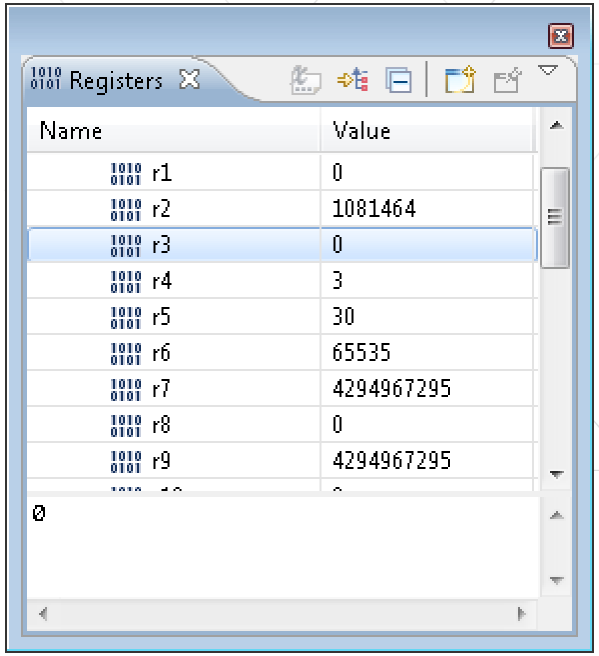
Address	0 - 3	4 - 7	8 - B	C - F
00000A60	F86018F8	B800FF84	3021FFE0	FA61001C
00000A70	F9E10000	BE05009C	12650000	E8650004
00000A80	F80018F8	B0001111	AA431111	BC120030
00000A90	30A014E0	B9F40374	30C000CF	30600001
00000AA0	10A00000	F86018F8	10650000	E9E10000
00000AB0	EA61001C	B60F0008	30210020	AA460001
00000AC0	BC120034	AA460002	BC120020	30A014E0

Debug Functionality

- Yellow represents registers that have changed (useful when following assembly code)
- To change any value, click to edit, or right-click the field



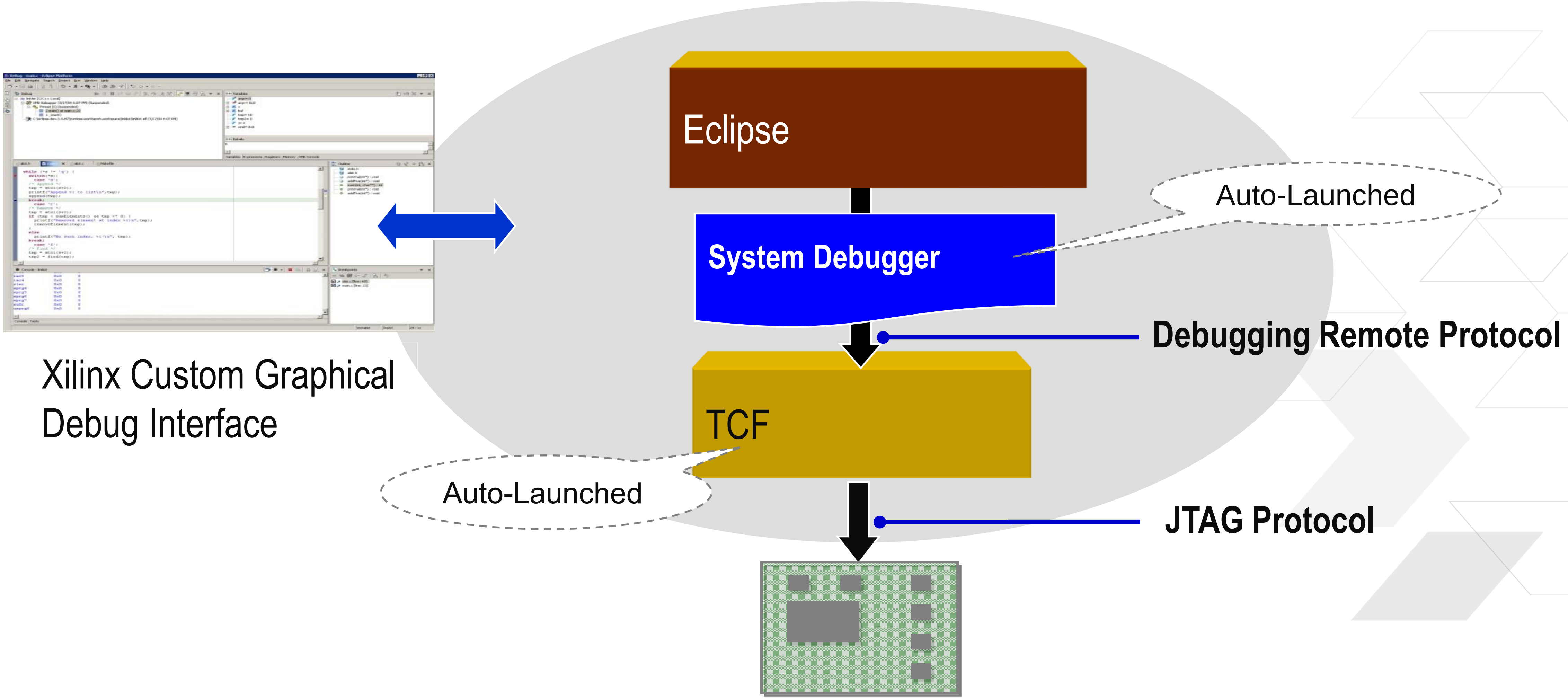




Outline

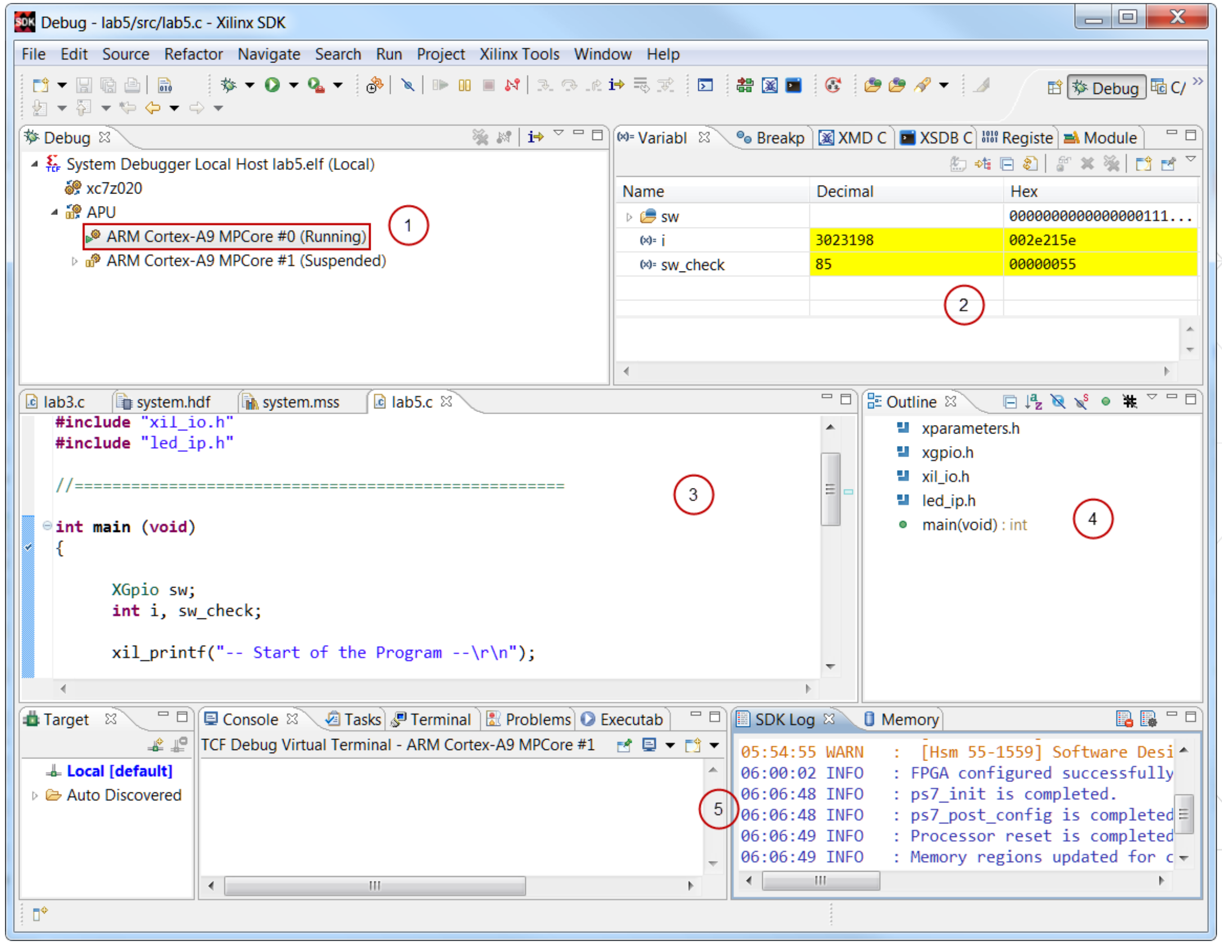
- Device Drivers Architecture
- Timers and API
- Debugging Tools
  - Hardware Tools
  - Software Tools
- ***Debug in SDK***
- Summary

Debugging Using SDK (TCF)



SDK Debug Perspective

- > **Stack frame for target threads**
- > **Variables, breakpoints, and registers views**
- > **C/C++ editor**
- > **Code outline**
  - >> Disassembly view can be added using Window > Show View > Disassembly
- > **Console, SDK Log, and Memory views**



XILINX

Outline

- > Device Drivers Architecture
- > Timers and API
- > Debugging Tools
  - >> Hardware Tools
  - >> Software Tools
- > Debug in SDK
- > ***Summary***

Summary

- Debugging is an integral part of embedded systems development
- Vivado provides tools to facilitate hardware and software debugging
  - Hardware debugging is done through using Vivado Logic Analyzer
  - Software debugging is performed using xmd and GNU debugger
- SDK provides environment, perspective, and underlying tools to enable seamless software debugging
- XMD/GDB Debug
  - Single service on dedicated connection
- System Debugger/TCF
  - Multicore Debug, shared connection

Questions?