Software Development and Debug

Zynq
14.2 Version

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

Outline

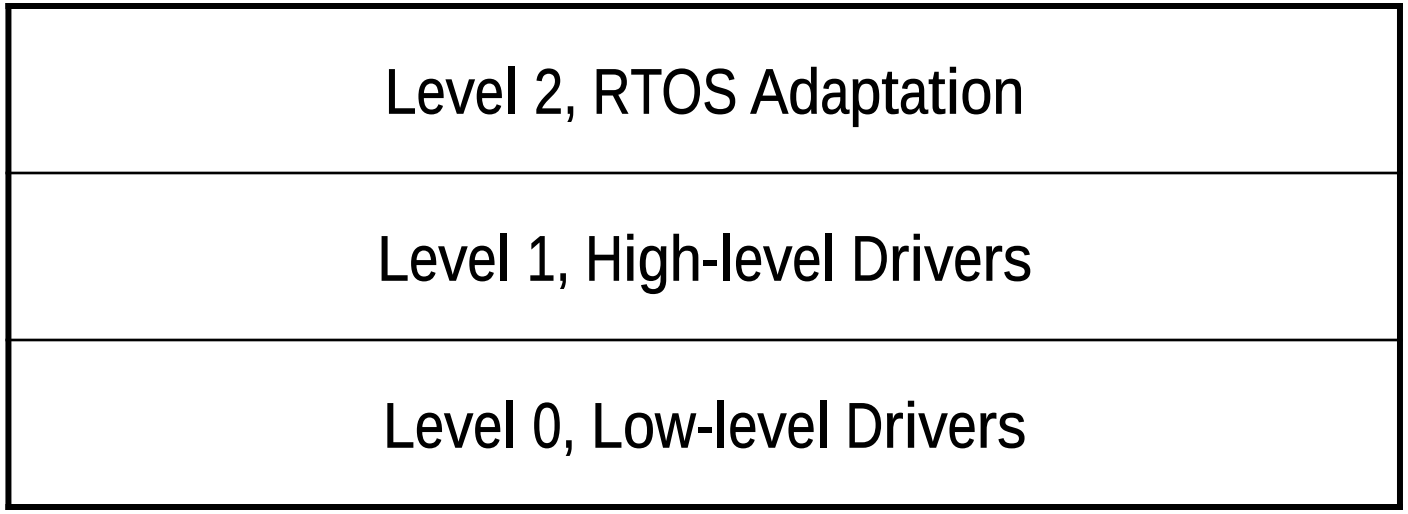
- ***Device Drivers Architecture***
- **Timers and API**
- **Debugging Tools**
  - Hardware Tools
  - Software Tools
- **Debug in SDK**
- **Summary**

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



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

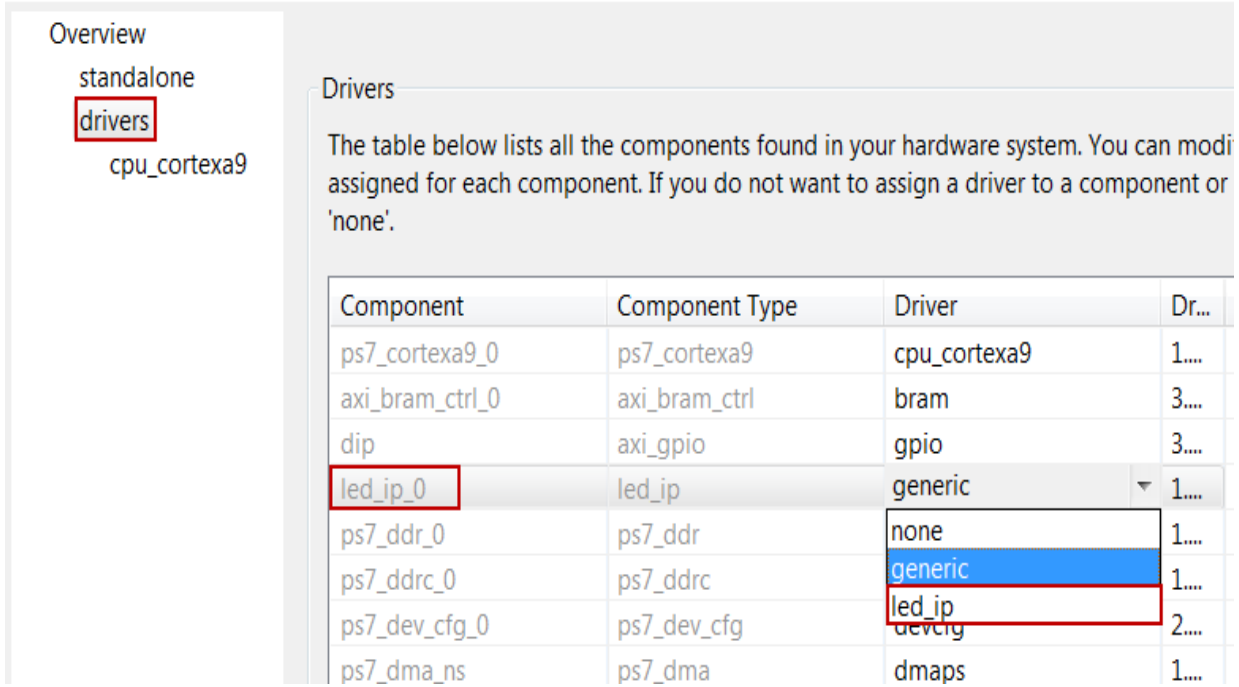
- **XUartPs_CfgInitialize() - Initializes a specific XUartPs instance such that it is ready to be used**
- **XUartPs_Send() - Sends the specified buffer using the device in either polled or interrupt driven mode.**
- **XUartPs_Recv() - Receive a specified number of bytes of data from the device and store it into the specified buffer.**
- **XUartPs_SetBaudRate() - Sets the baud rate for the device.**

UARTPS Level 0

- **XUartPs_SendByte()- Sends one byte using the device.**
- **XUartPs_RecvByte()- Receives a byte from the device.**

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

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 match the 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

Software Debugging Support

- **EDK/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

Hardware Debugging Support

- **EDK supports hardware debugging via the following tools**
  - ChipScope Pro software
    - Soft-core base logic analyzer
    - Operates through a Xilinx download cable
- **Processor software simulator**
  - Part of XMD
  - Cycle-accurate simulation
  - Memory access only; no bus peripherals
- **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
  - Program download

XMD Debugger

- For debugging standalone or bare-metal applications, serves as gdbserver for gdb and SDK
- For Linux applications, SDK interacts with a gdbserver running on the target
- XMD is started via the Xilinx Tools > Program FPGA command in SDK
- XMD is essentially launched and controlled via the RUN menu

XMD Functionality

- **XMD engine**
  - Program that facilitates a unified GDB interface
  - Tcl interface and command interpreter
- **XMD supports application debugging on different targets**
- **GDB 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

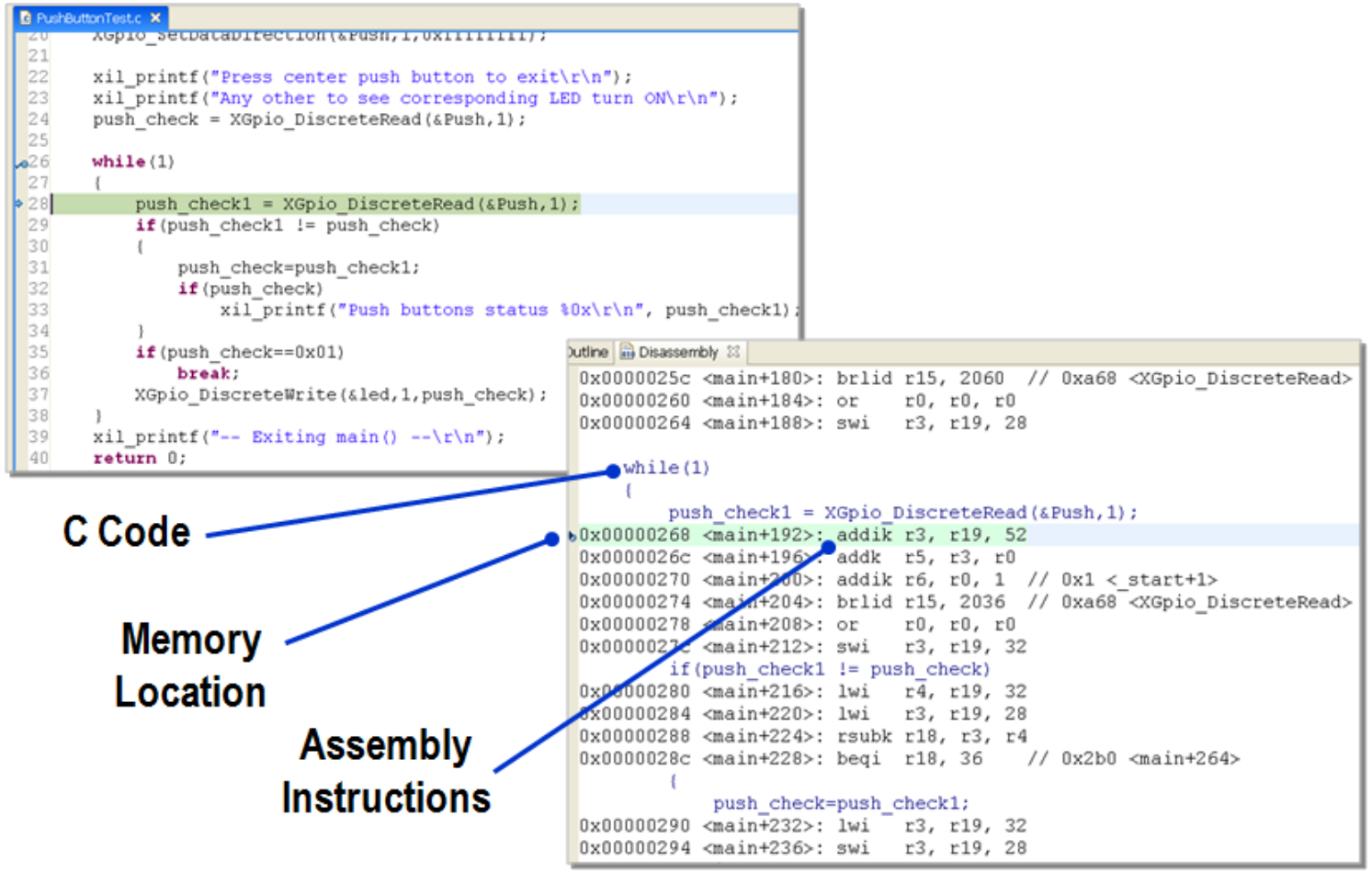
XMD Tcl Inteface

- 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

GDB Functionality

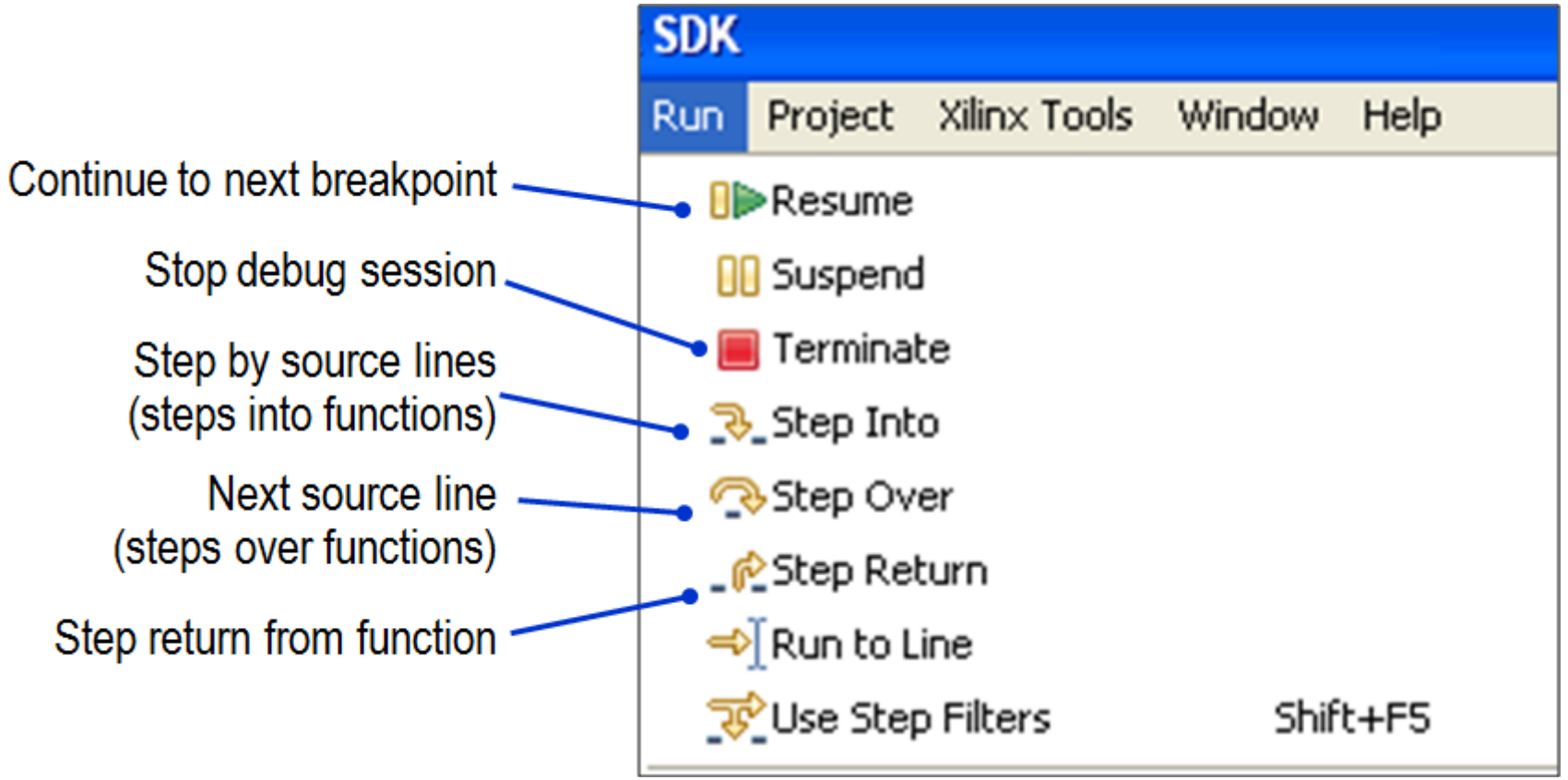
- **GDB is a source-level debugger that helps you debug your program**
  - Start your program
  - Set breakpoints (make your program stop on specified conditions)
  - Examine what has happened, when your program encounters breakpoints
    - Registers
    - Memory
    - Stack
    - Variables
    - Expressions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

GDB



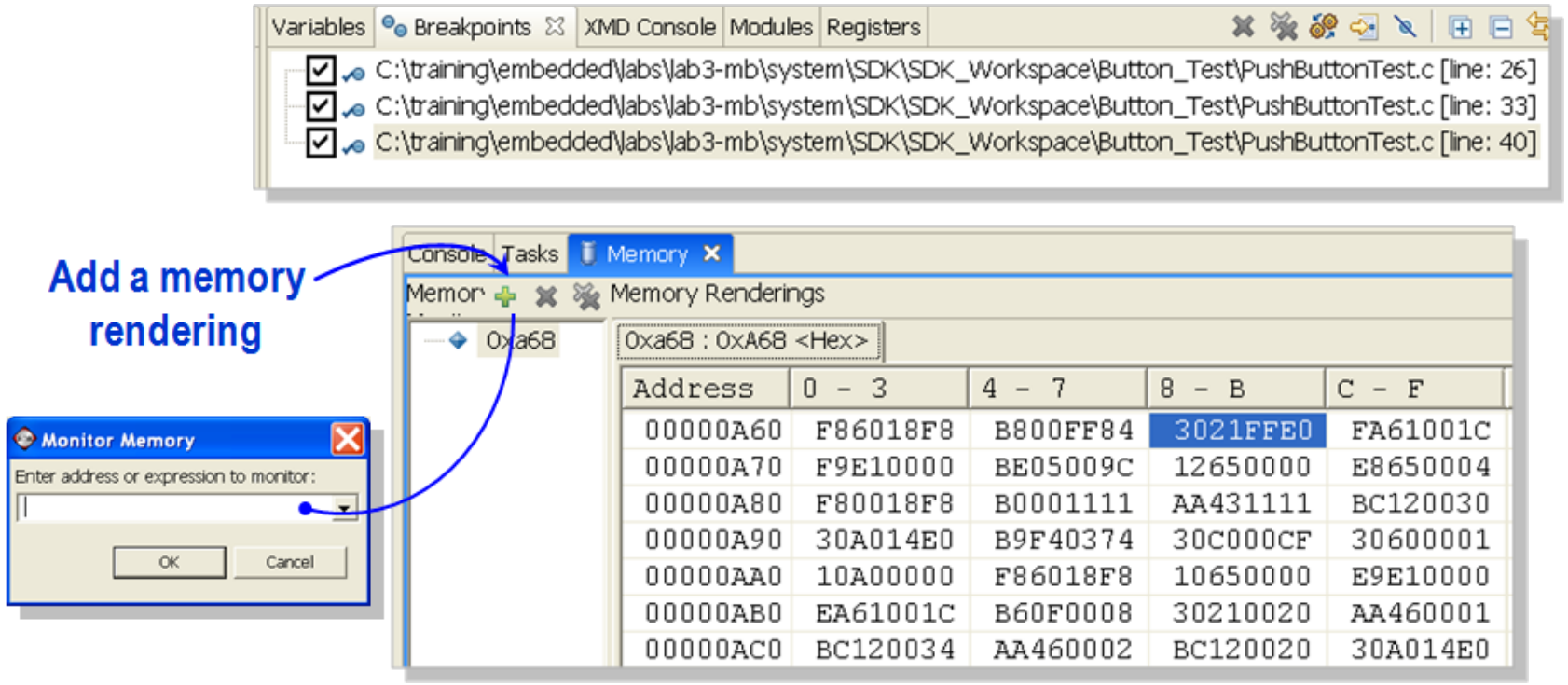
GDB GUI

- **Run-time control**



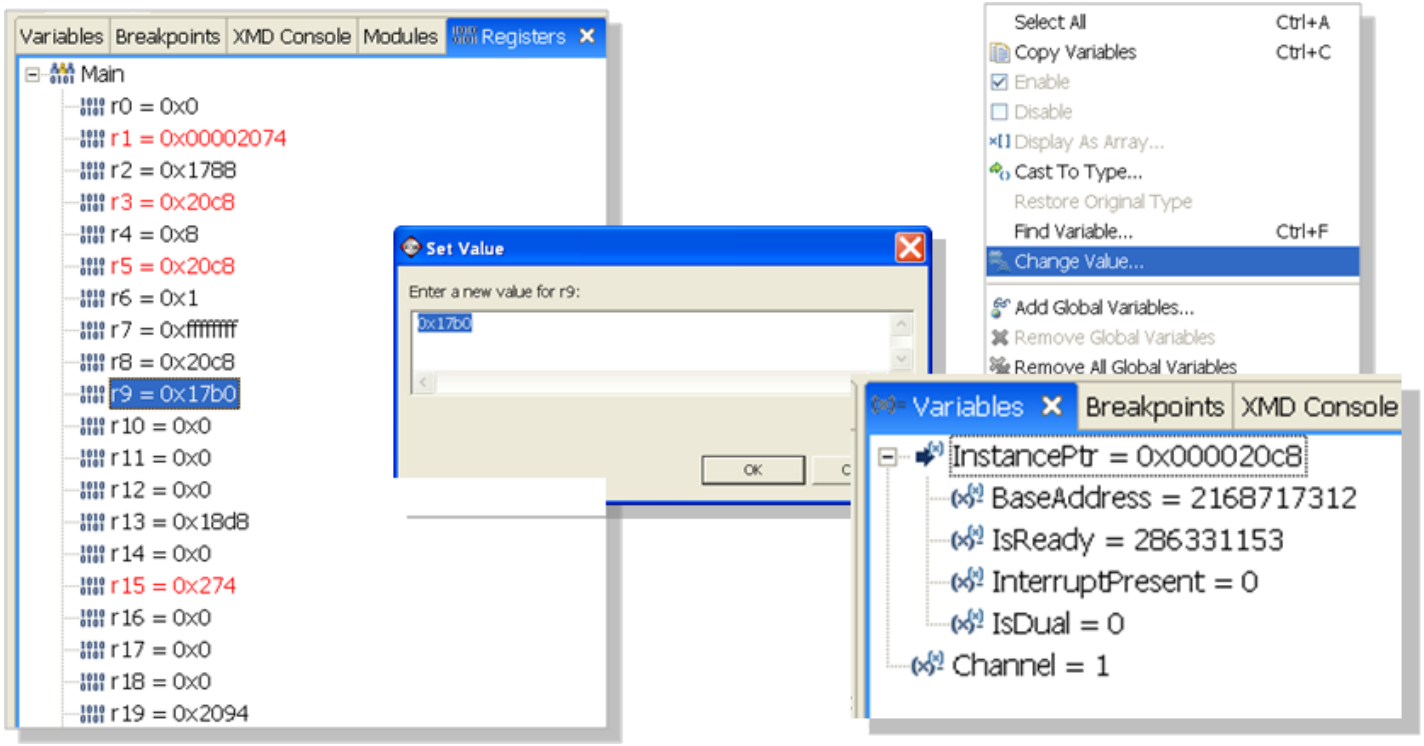
GDB Functionality

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



GDB Functionality

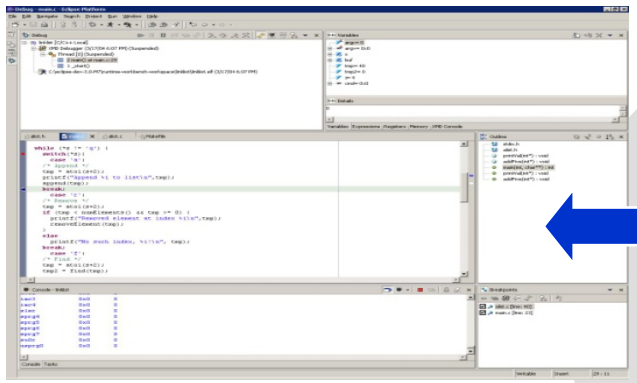
- Blue represents registers that have changed (useful when following the assembly code)
- To change any value, right-click the field



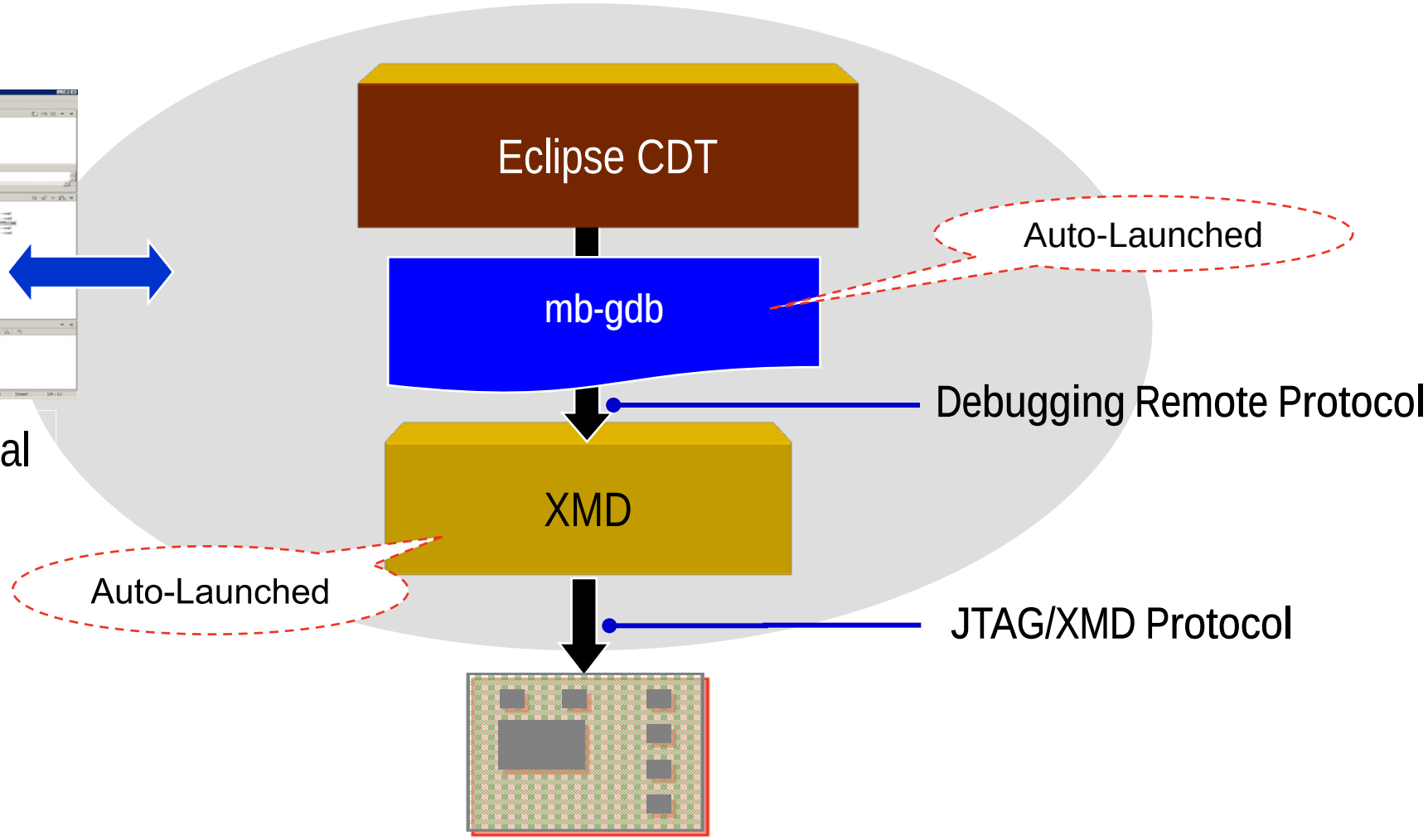
Outline

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

Debugging Using SDK

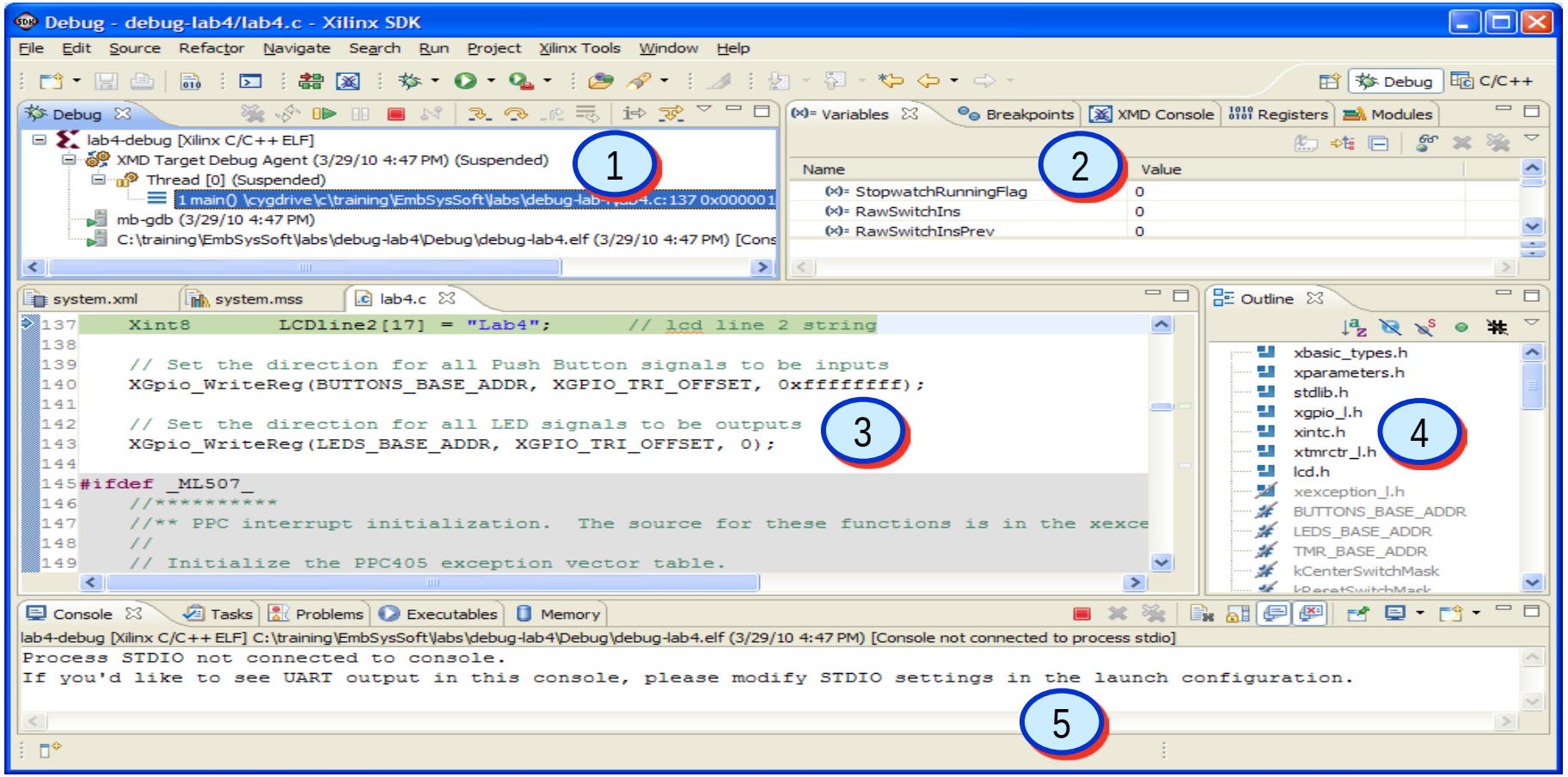


Xilinx Custom Graphical Debug Interface



SDK Debug Perspective

1. Stack frame for target threads
2. Variables, breakpoints, and registers views
3. C/C++ editor
4. Code outline and disassembly view
5. Console view



Outline

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

Summary

- **Debugging is an integral part of embedded systems development**
- **EDK provides tools to facilitate hardware and software debugging**
  - Hardware debugging is done through using ChipScope cores and ChipScope Analyzer
  - Software debugging is performed using xmd and GNU debugger
- **SDK provides environment, perspective, and underlying tools to enable seamless software debugging**
- **XMD debugger provides**
  - Download cable connection to a processor target
  - TCP/IP communications port
  - Set of primitive debugs commands with a Tcl interface
- **XMD can be launched from the XMD console or a Bash shell**
- **GDB debugger provides a debug IDE that issues primitive debug commands to XMD**
- **XMD and GDB provide multiple-session processor support**