Software Development Environment

Zynq
Vivado 2018.3 Version

Objectives

- **After completing this module, you will be able to:**
  - Understand the basic concepts of the Eclipse IDE in SDK
  - List SDK features
  - Identify the GNU tools functionality
  - List steps in creating a software application
  - State when address management is needed
  - Describe the object file sections
  - Describe what a linker script does

Outline

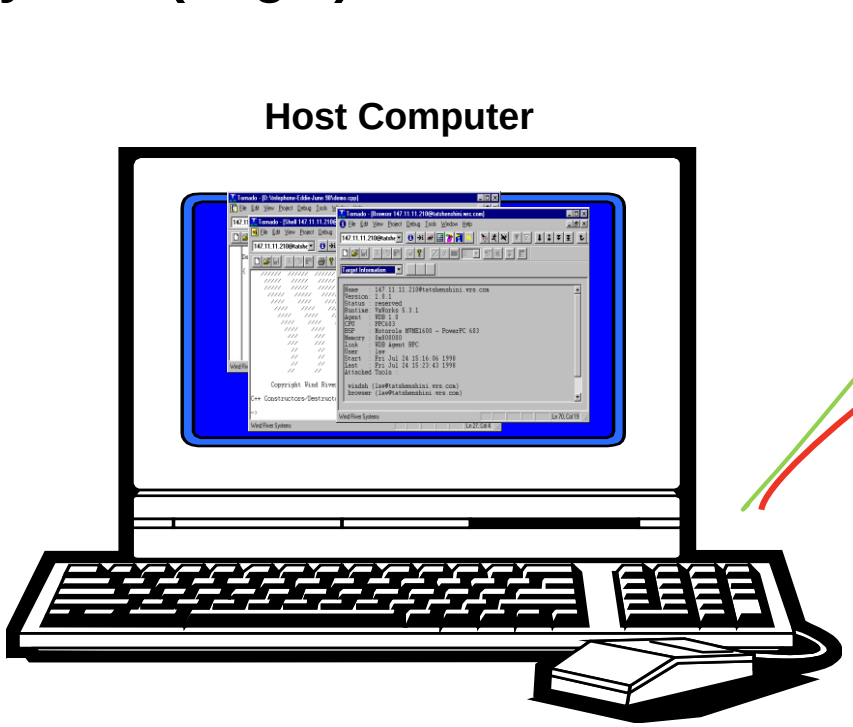
- ***Introduction***
- **SDK Development Environment**
- **SDK Project Creation**
- **GNU Development Tools: GCC, AS, LD, Binutils**
- **Software Settings**
  - Software Platform Settings
  - Compiler Settings
- **Address Management**
- **Object File Sections**
- **Linker Script**
- **Summary**

Desktop versus Embedded

- **Desktop development: written, debugged, and run on the same machine**
- **OS loads the program into the memory when the program has been requested to run**
- **Address resolution takes place at the time of loading by a program called the loader**
  - The loader is included in the OS
- **The programmer glues into one executable file called ELF**
  - Boot code, application code, RTOS, and ISRs
  - Address resolution takes place during the *gluing* stage
- **The executable file is downloaded into the target system through different methods**
  - Ethernet, serial, JTAG, BDM, ROM programmer

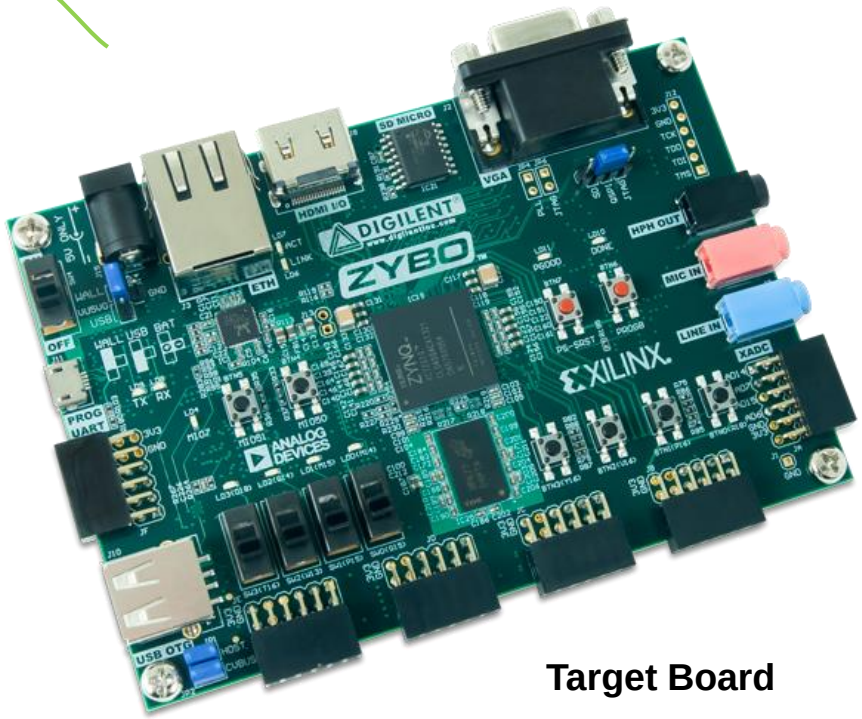
Embedded versus Desktop

- **Development takes place on one machine (host) and is downloaded to the embedded system (target)**



JTAG

UART



Target Board

A cross-compiler is run on the host

Outline

- Introduction
- ***SDK Development Environment***
- SDK Project Creation
- GNU Development Tools: GCC, AS, LD, Binutils
- Software Settings
  - Software Platform Settings
  - Compiler Settings
- Address Management
- Object File Sections
- Linker Script
- Summary

Eclipse/CDT Frameworks of SDK

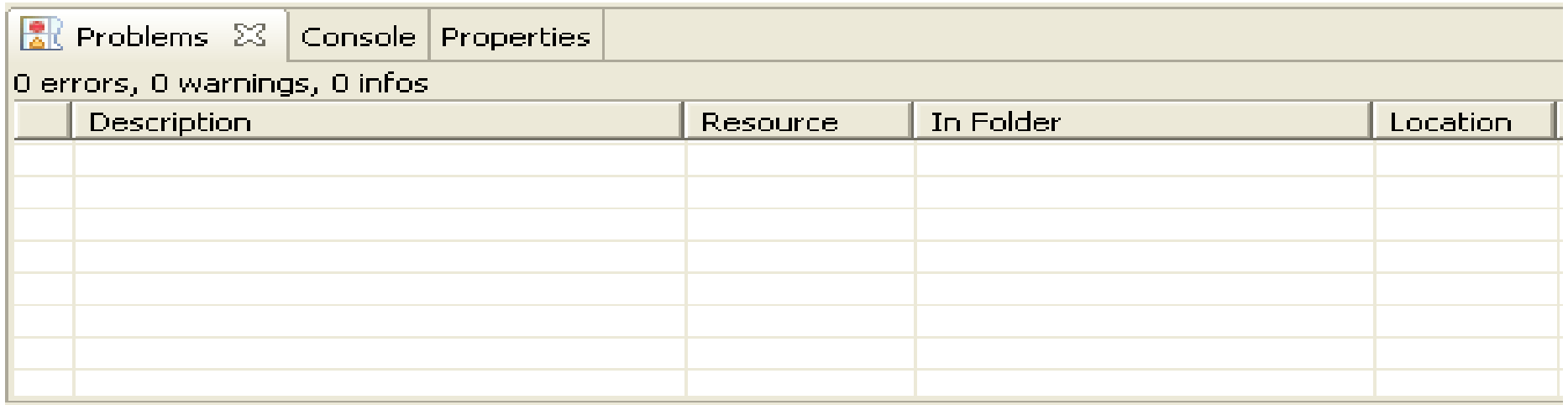
- **Builder framework**
  - Compiles and Links Source files
  - Default Build options are specified when application is created: Choice of Debug, Release, Profile configurations
  - User can custom build options later when developing application
  - Build types: Standard Make, Managed Make
- **Launch framework**
  - Specifies what action needs to be taken: Run (+ Profile) application or Debug application
  - In SDK, this is akin to the Target Connection settings
- **Debug framework**
  - Launches debugger(gdb), loads application and begins debug session
  - Debug views show information about state of debug session
- **Search framework**
  - Helps development of application
- **Help System**
  - Online help system; context-sensitive

Workspaces and Perspectives

- **Workspace**
  - Location to store preferences & internal info about projects
  - Transparent to users
  - Source files not stored under Workspace
- **Views, Editors**
  - Basic user interface element
- **Perspectives**
  - Collection of functionally related views
  - Layout of views in a perspective can be customized according to user preference

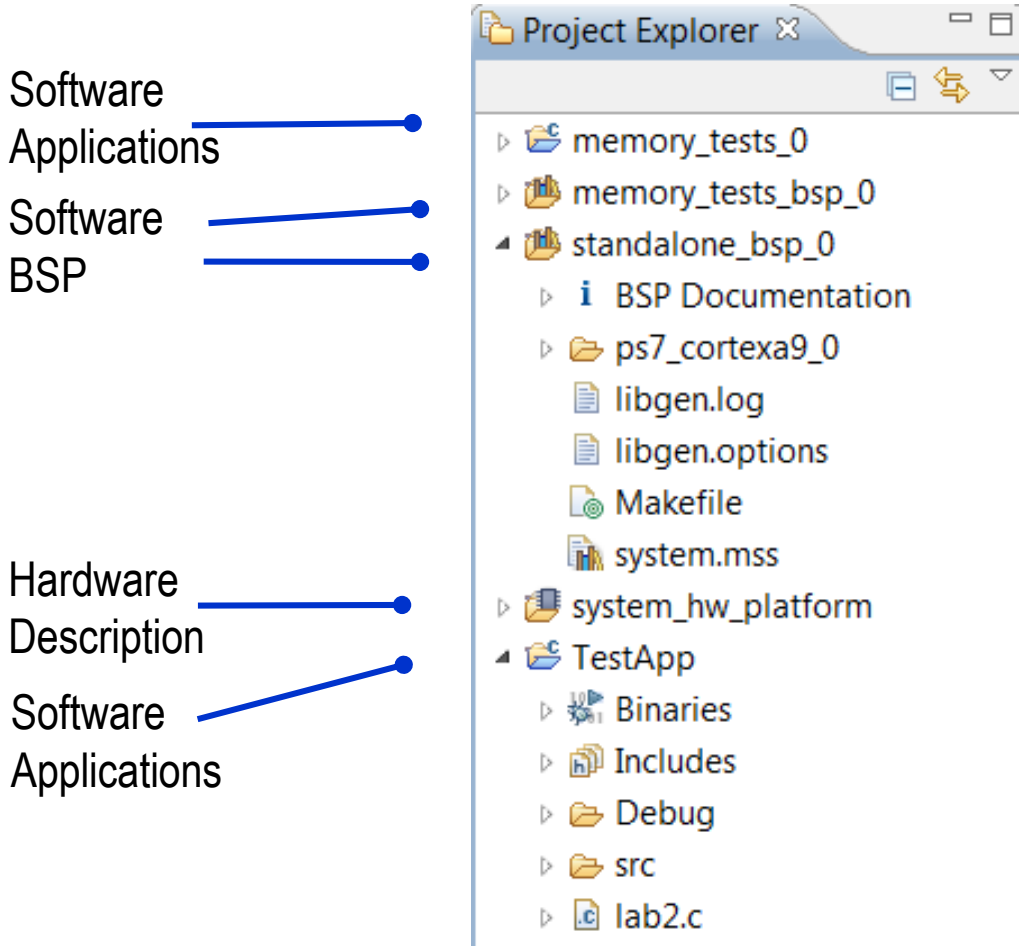
Views

- Eclipse Platform views: Navigator view, Tasks view, Problems view
- Debug views: Stack view, Variables view
- C/C++ views: Projects view, Outline view



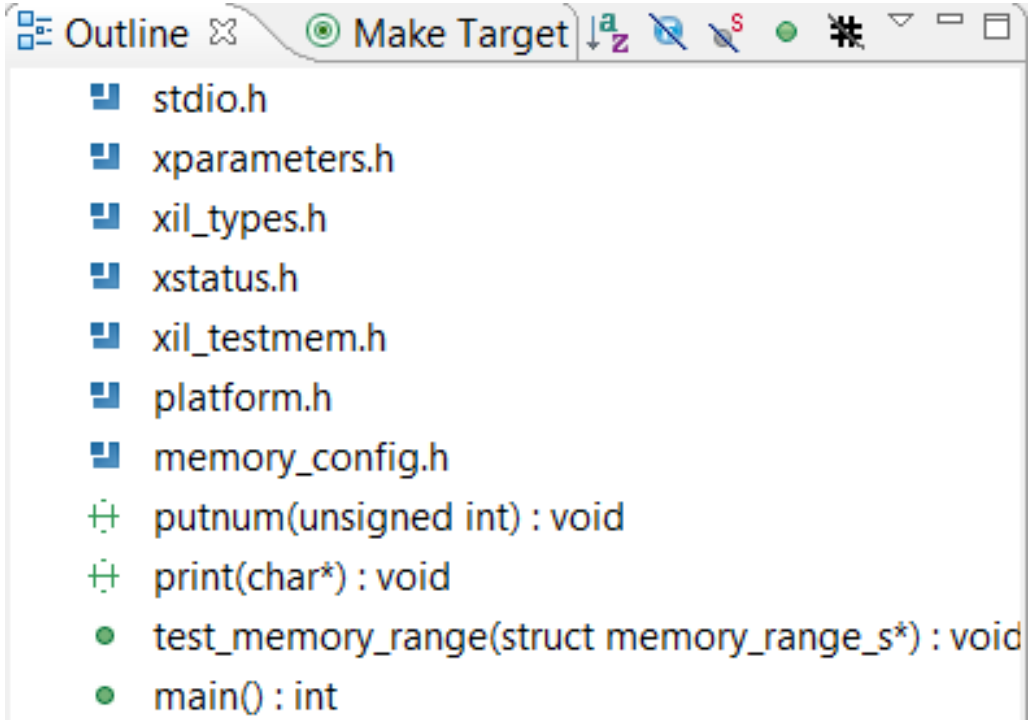
C/C++ Project View

- **Hierarchical list of the workspace projects in a hierarchical format**
- **Double-click to open a file**
- **Right-click the project to access its properties**



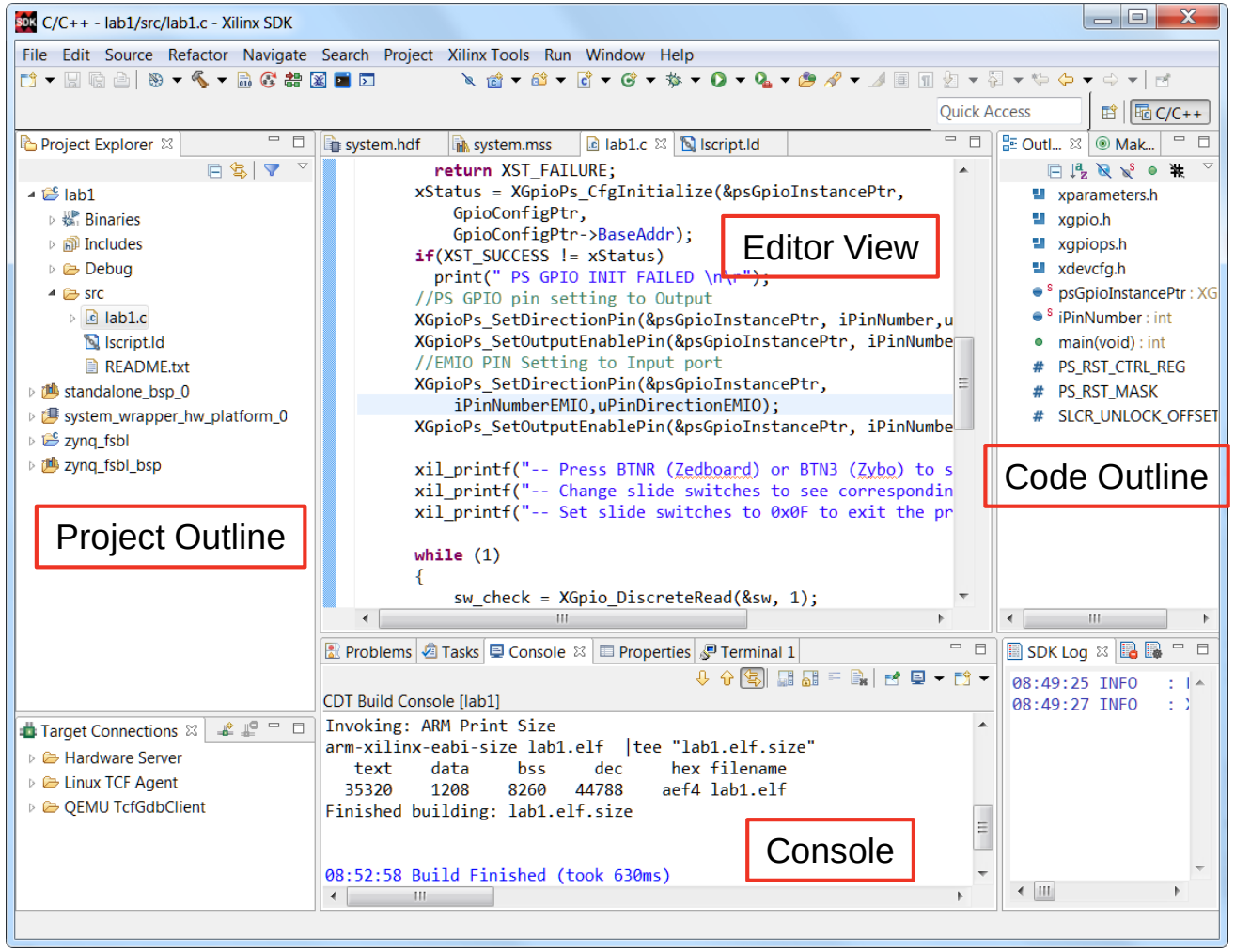
Outline View

- **Displays an outline of the structured file that is currently open in the editor**
- **The contents of the outline view are editor specific**
- **Content type is indicated by the icon**
- **For a C source, icons represent**
  - *#define* statements
  - Include files
  - Function calls
  - Declarations
- **Selecting a symbol will navigate to the same in the editor window**



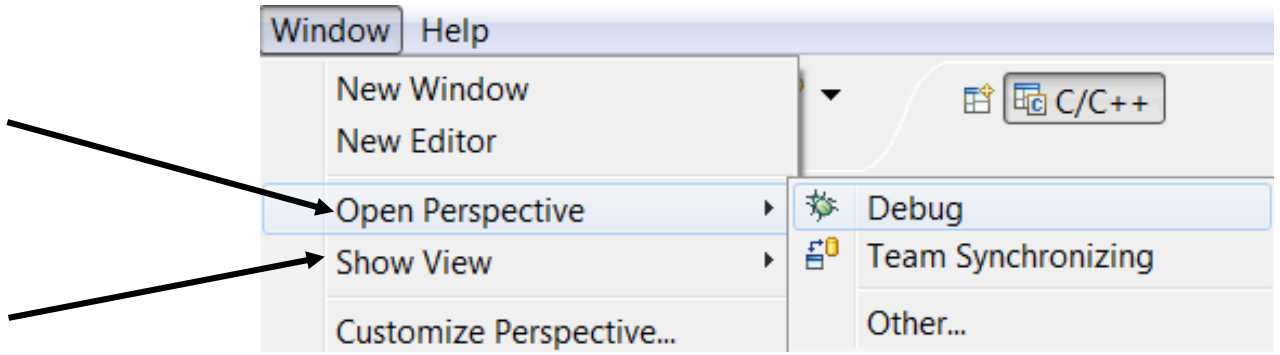
C/C++ Perspective

- C/C++ project outline displays the elements of a project with file decorators (icons) for easy identification
- C/C++ editor for integrated software creation
- Code outline displays elements of the software file under development with file decorators (icons) for easy identification
- Problems, Console, Properties view lists output information associated with the software development flow



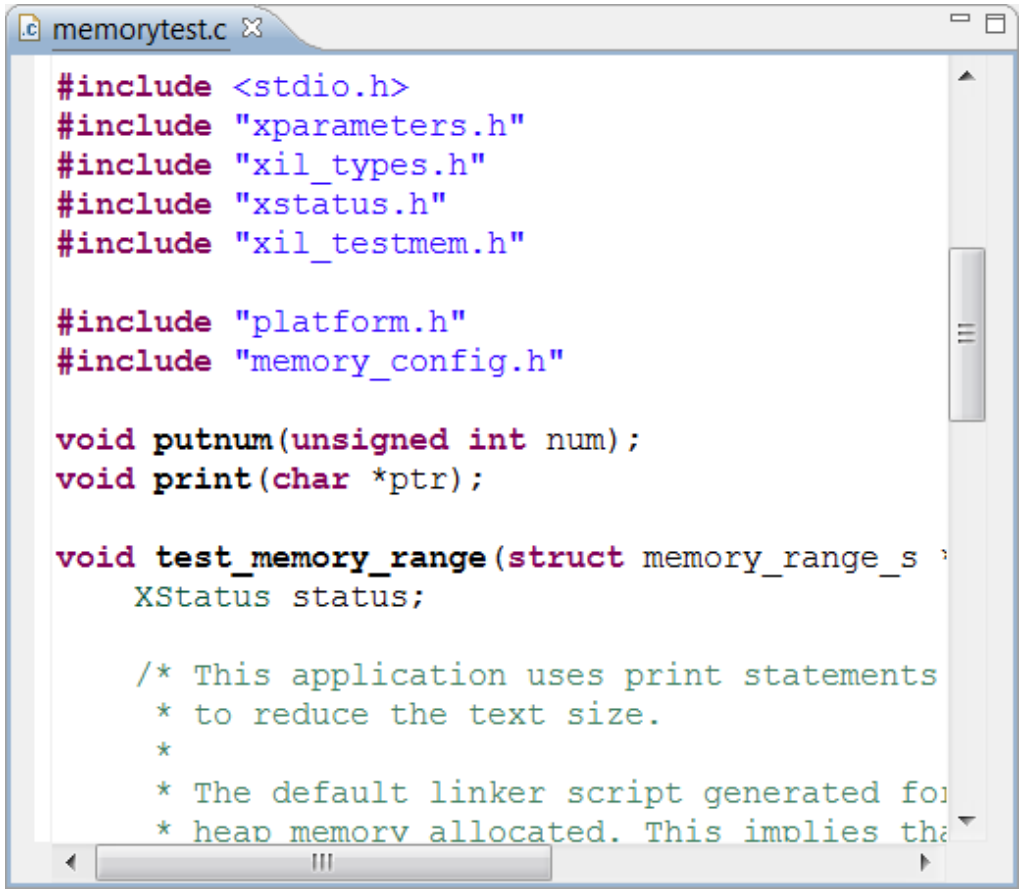
Opening Perspectives and Views

- **To open a Perspective, use**
  - **Window → Open Perspective**
- **To open a view, use**
  - **Window → Show View**
  - If the view is already present in the current perspective, the view is highlighted



Editor

- **Syntax Highlighting**
  - bracket matching
  - syntax coloring
  - content assist
  - refactoring
  - keyboard shortcuts



Outline

- Introduction
- SDK Development Environment
- ***SDK Project Creation***
- GNU Development Tools: GCC, AS, LD, Binutils
- Software Settings
  - Software Platform Settings
  - Compiler Settings
- Address Management
- Object File Sections
- Linker Script
- Summary

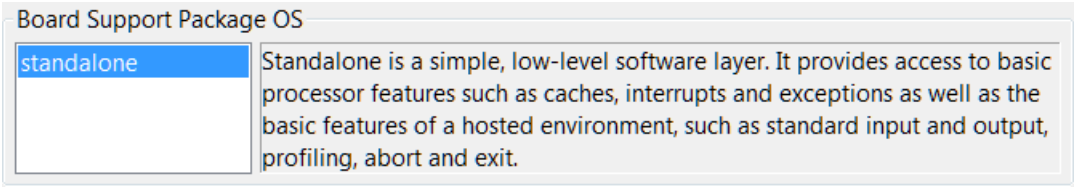
Launching SDK

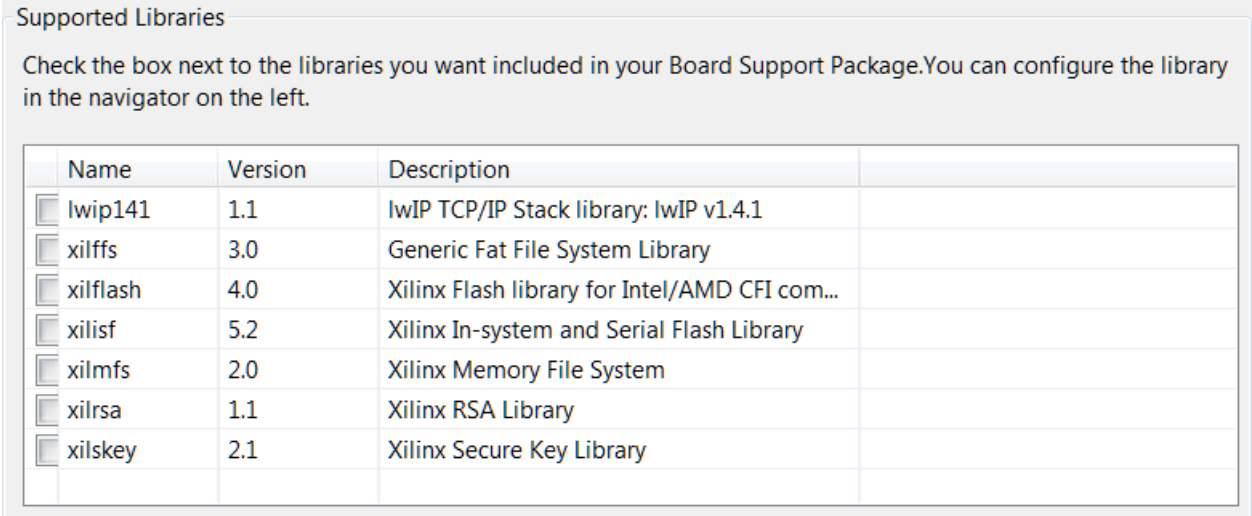
Launch SDK

- Standalone
  - Choose workspace, choose Hardware Platform Specification
- In Vivado
  - **File> Export Hardware**
  - **File > Launch SDK**
- Exporting
  - A Hardware Description File HDF file is first generated
  - A hardware platform specification project is then automatically created
    - The software application (and board support package) then can be created and associated with the hardware platform

Creating a Board Support Package

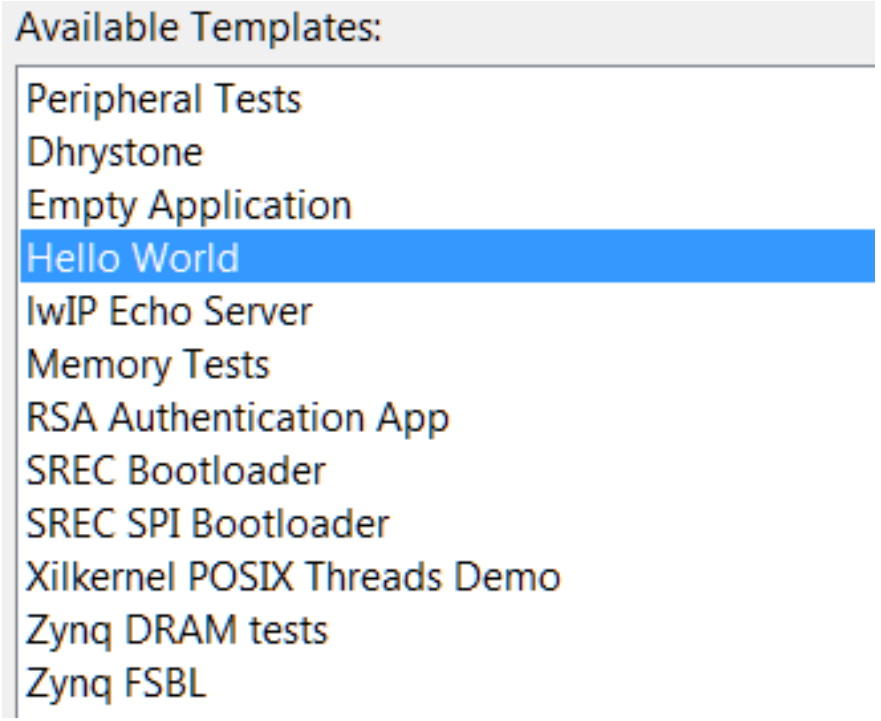
- **The Board Support Package provides software services based on the processor and peripherals that make up the processor system**
- **Can be automatically created when creating Application project**
- **Can be created standalone**
- **Must be attached to a Hardware Platform**
  - **File > New > Board Support Package**
  - Select appropriate OS support
  - Third-party operating systems are supported with the appropriate BSP selection
  - Select required libraries support





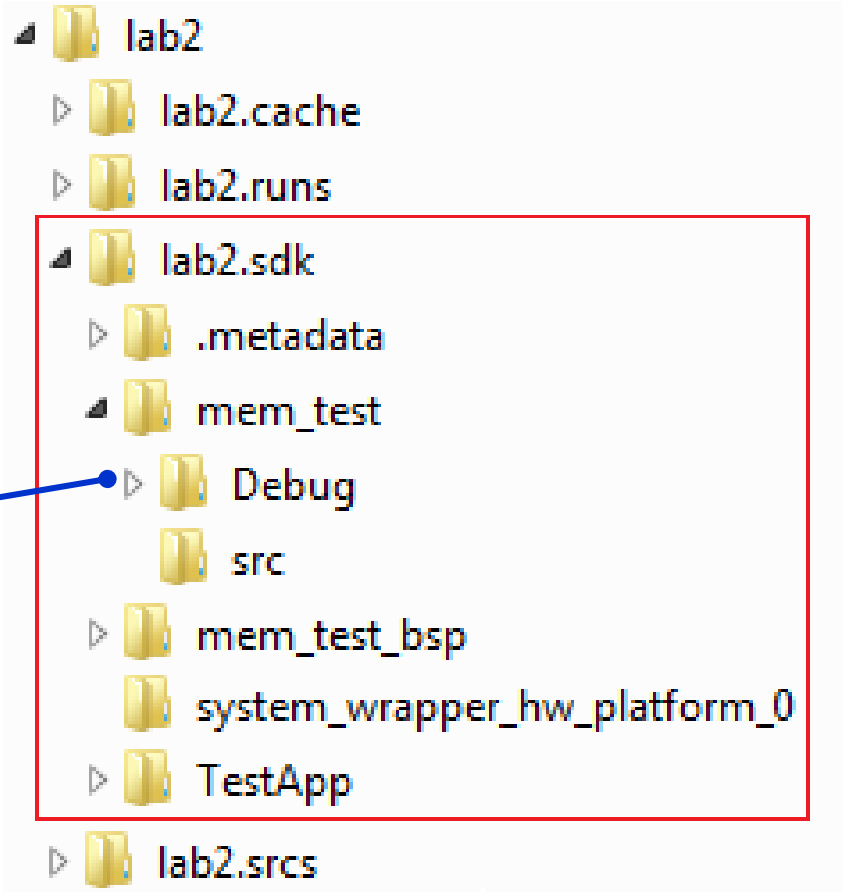
Creating a Software Application Project

- **SDK supports multiple software application projects**
- **A software project is attached to a BSP project**
- **Sample applications are provided**
  - Great for quick test of hardware
    - Peripheral Tests
  - Starting point to base your own application on
- **Typically an Empty Application is opened to begin a non-standard project**



Directory Structure

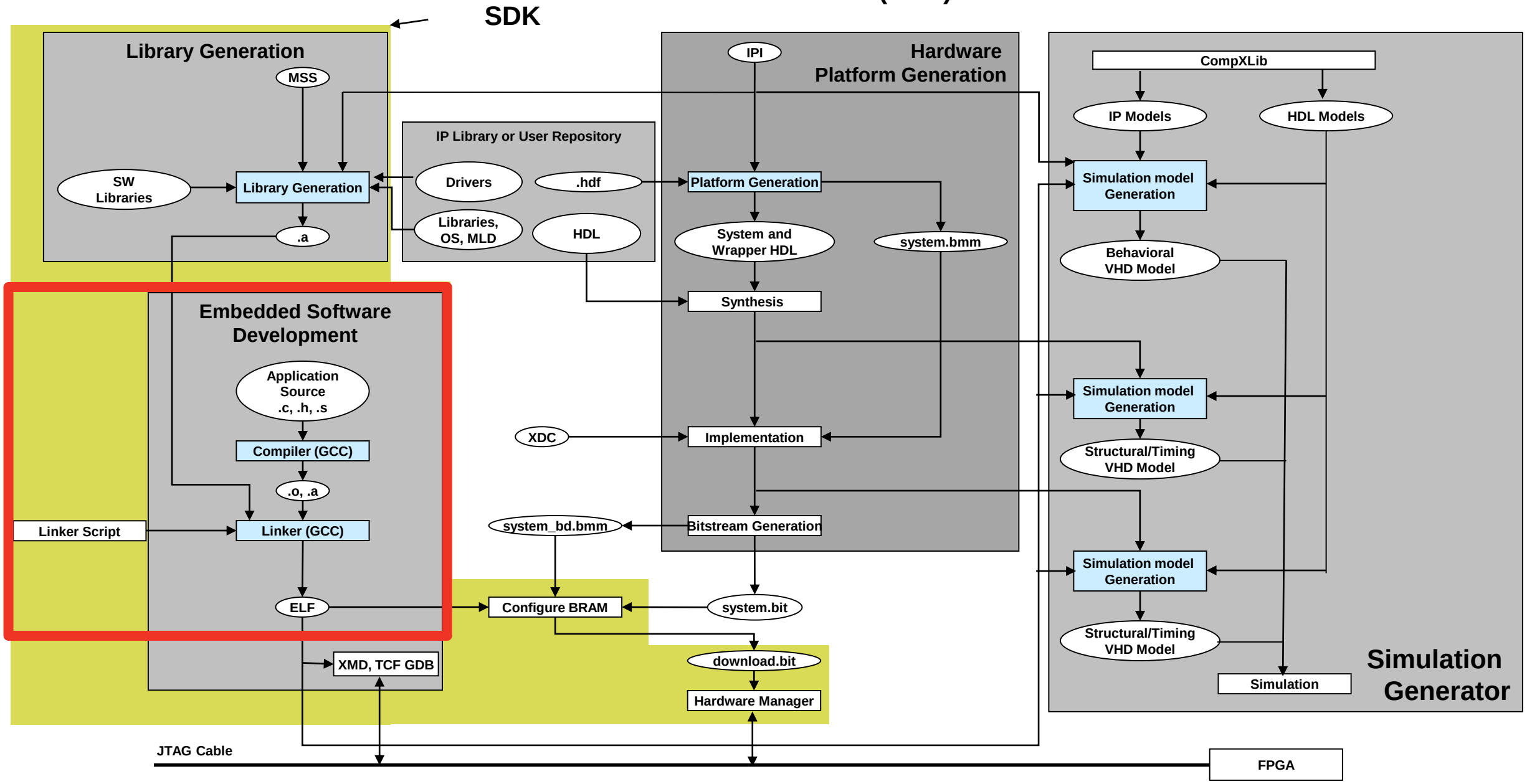
- **SDK projects are place in the application directory that was specified when SDK was launched**
- **Each project may have multiple directories for system files and configurations**
- **Configurations are property tool option permutations of the software application. Each configuration has project properties set depending on needs. An ELF file is generated for each**
  - Release configuration
  - Debug configuration
  - Profile configuration
- **A Debug configuration is created by default**



Outline

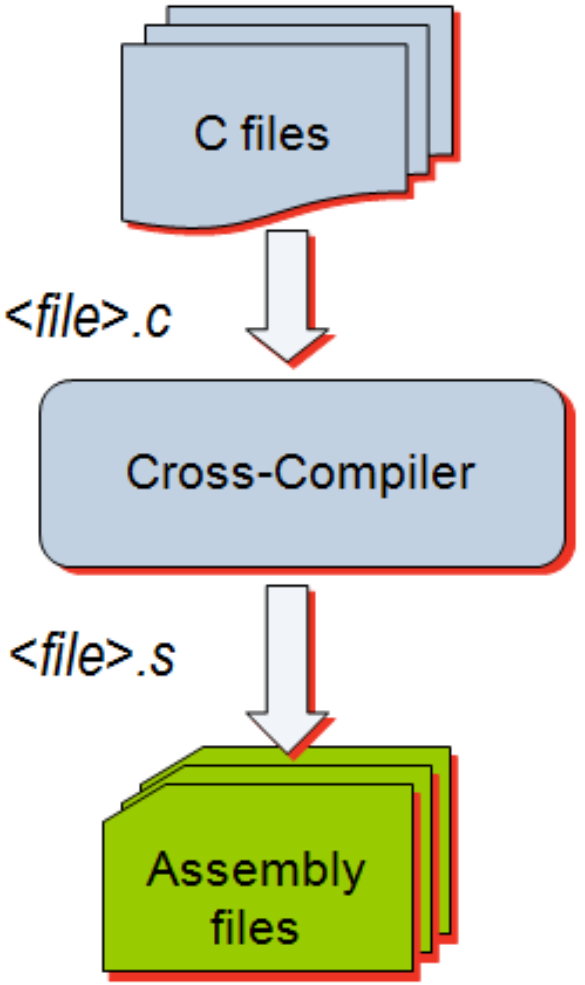
- Introduction
- SDK Development Environment
- SDK Project Creation
- ***GNU Development Tools: GCC, AS, LD, Binutils***
- Software Settings
  - Software Platform Settings
  - Compiler Settings
- Address Management
- Object File Sections
- Linker Script
- Summary

Embedded Tool Flow (SDK)



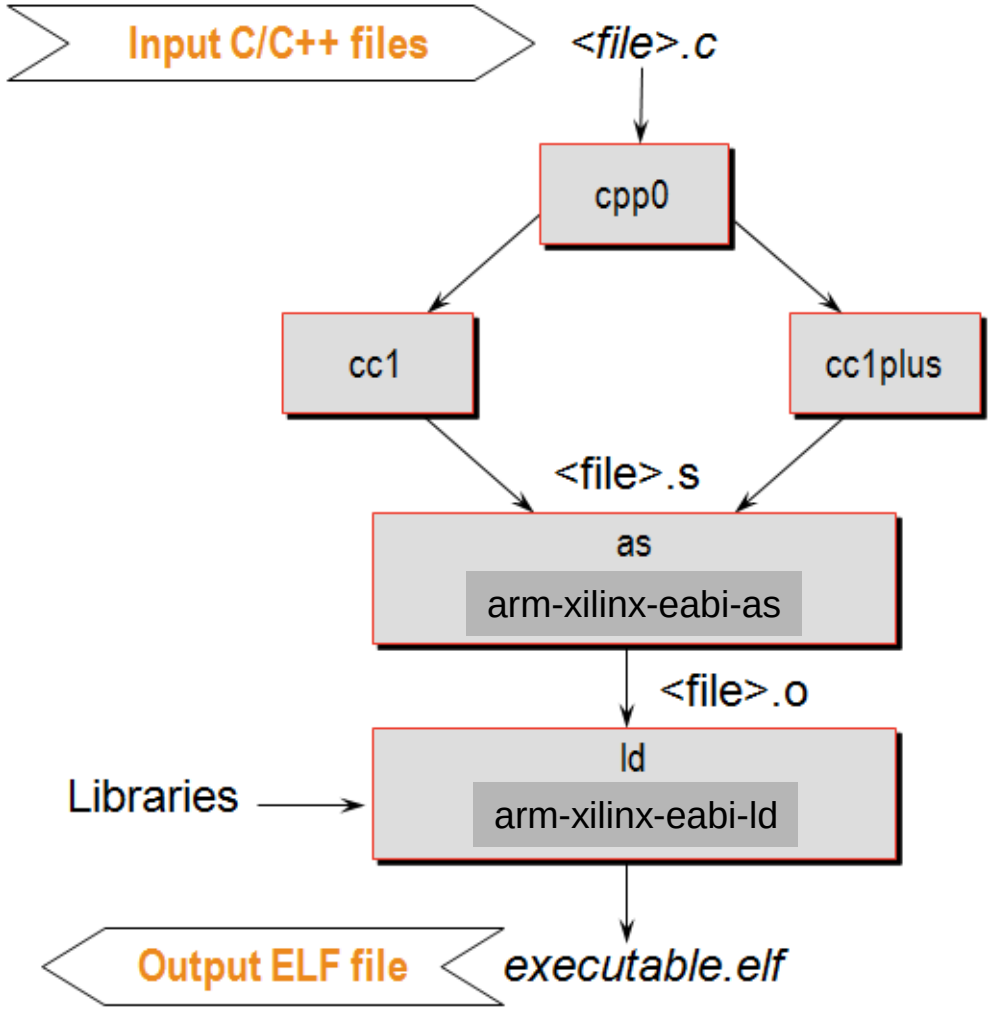
GNU Tools: GCC

- **GCC translates C source code into assembly language**
- **GCC also functions as the user interface to the GNU assembler and to the GNU linker, calling the assembler and the linker with the appropriate parameters**
- **Supported cross-compilers:**
  - GNU GCC (arm-xilinx-eabi-gcc)
- **Command line only; uses the settings set through the GUI**



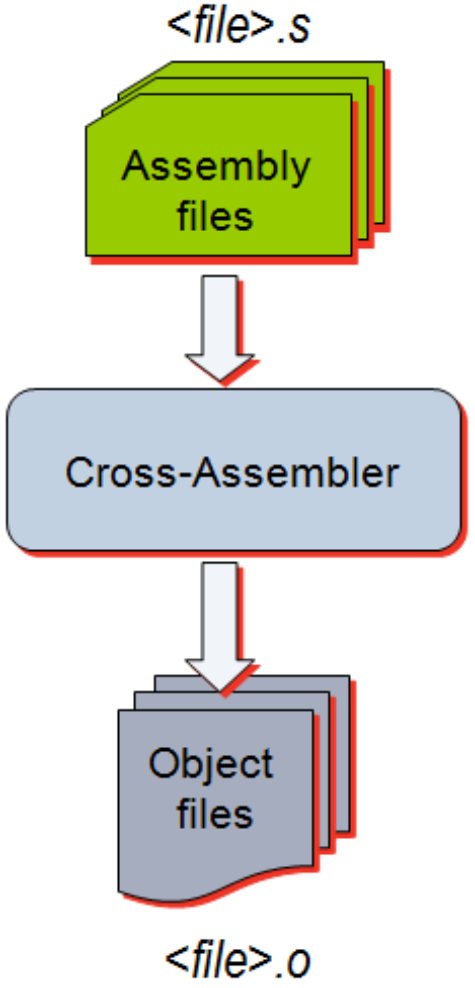
GNU Tools: GCC

- **Calls four different executables**
  - Preprocessor (cpp0)
    - Replaces all macros with definitions defined in the source and header files
  - Language specific c-compiler
    - cc1 C-programming language
    - cc1plus C++ language
  - Assembler
    - arm-xilinx-eabi-as
  - Linker
    - arm-xilinx-eabi-ld



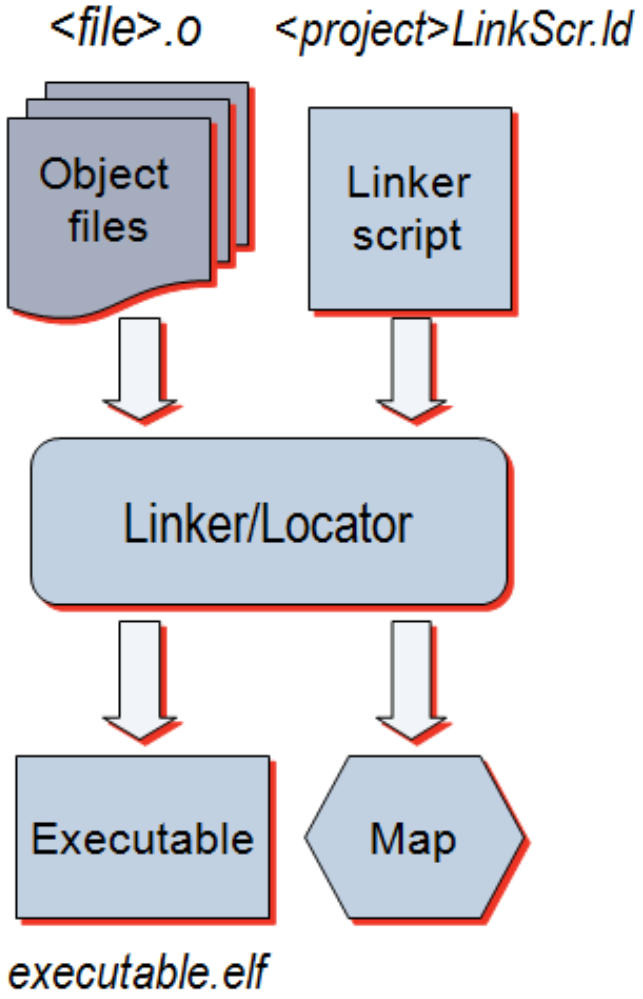
GNU Tools: AS

- **Input: Assembly language files**
  - File extension: **.s**
- **Output: Object code**
  - File extension: **.o**
  - Contains
    - Assembled piece of code
    - Constant data
    - External references
    - Debugging information
- **Typically, the compiler automatically calls the assembler**
- **Use the -Wa switch if the source files are assembly only and want to use the gcc**



GNU Tools: LD

- **Linker**
- **Inputs:**
  - Several object files
  - Archived object files (library)
  - Linker script (mapfile)
- **Output:**
  - Executable image (.ELF)
  - Map file

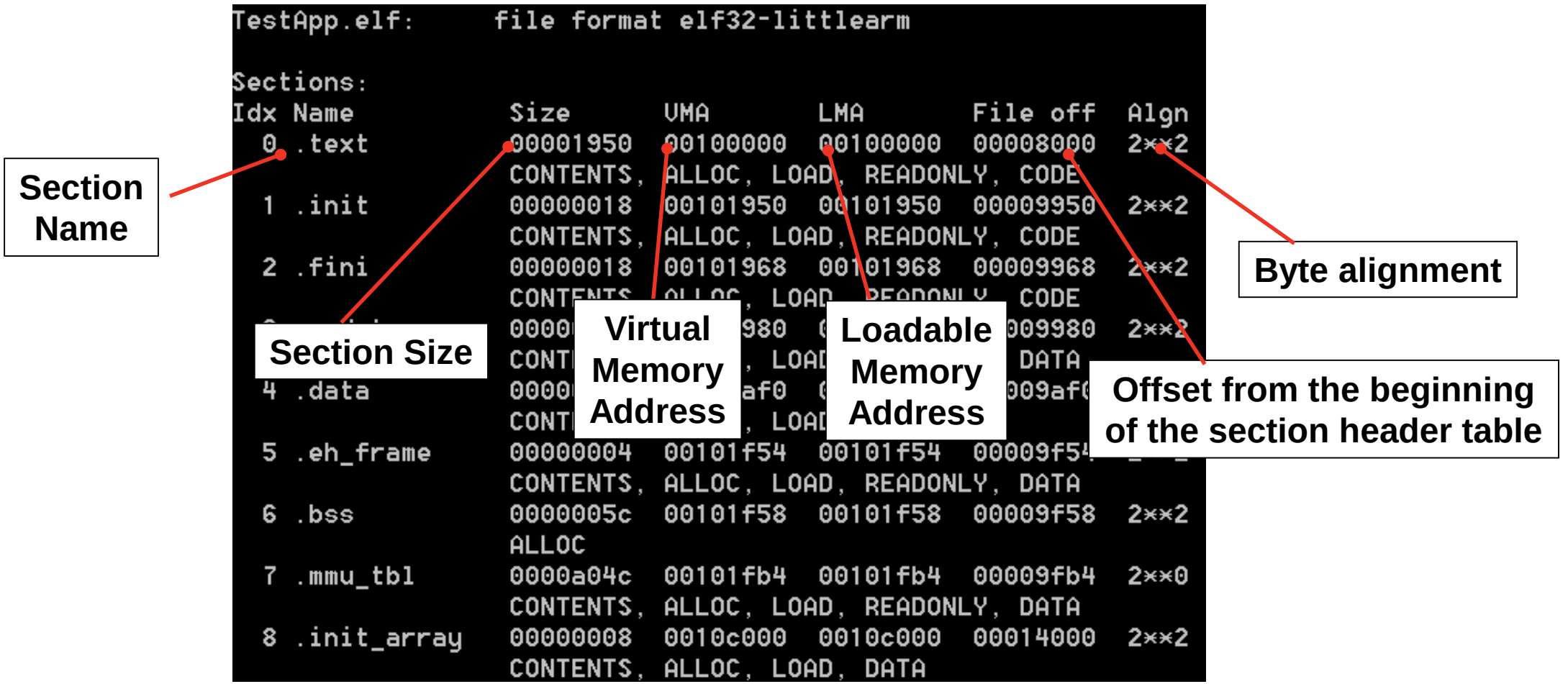


GNU Utilities

- **AR Archiver**
  - Create, modify, and extract from libraries
  - Used in SDK to combine the object files of the Board Support Package (BSP) in a library
  - Used in SDK to extract object files from different libraries
- **Object Dump**
  - Display information from object files and executables
    - Header information, memory map
    - Data
    - Disassemble code

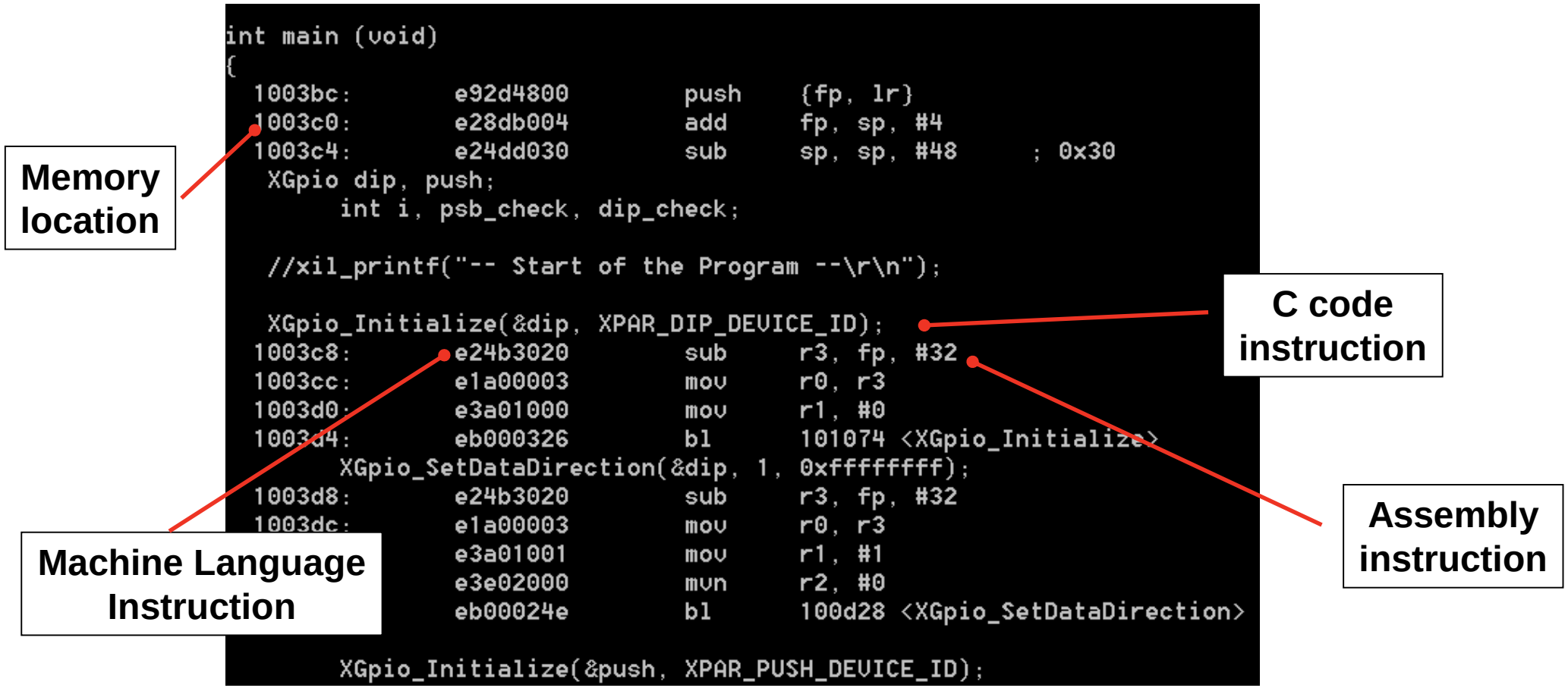
Object Dump
Display summary information from the section headers

arm-xilinx-eabi-objdump –h executable.elf



Object Dump
Dumping the source and assembly code

arm-xilinx-eabi-objdump –S executable.elf



Outline

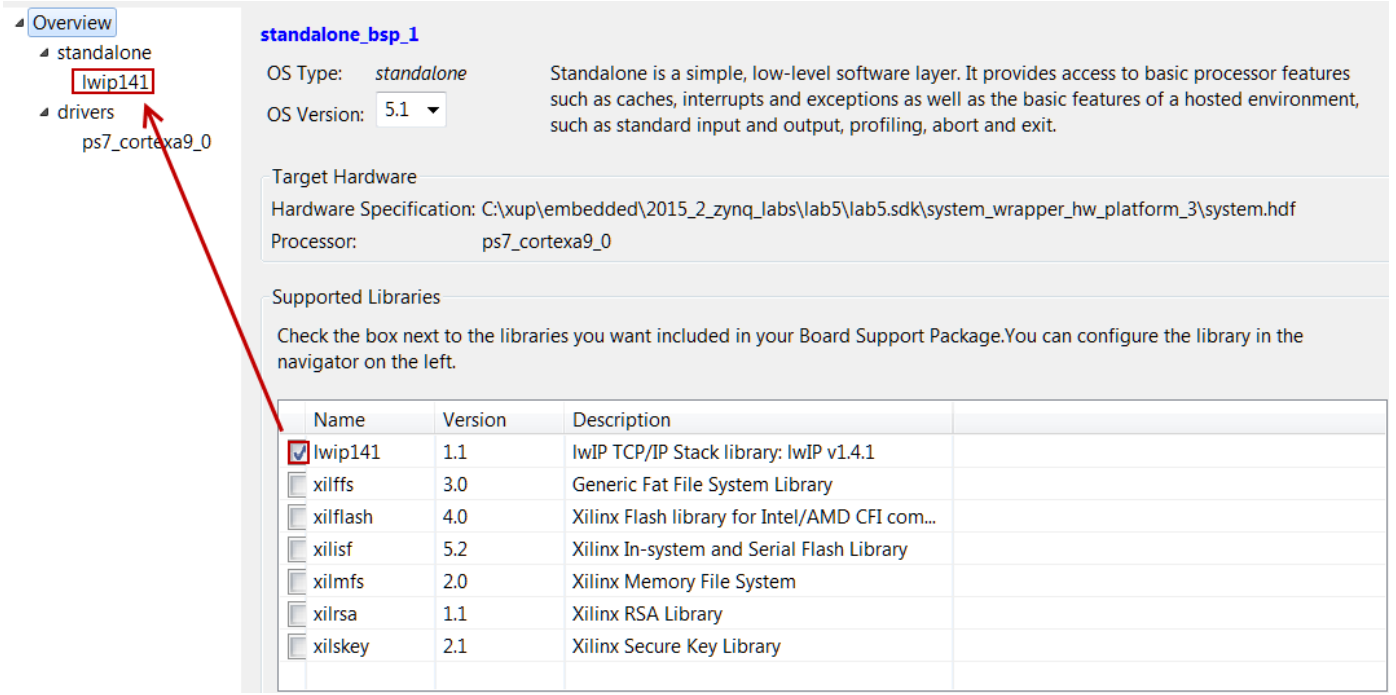
- Introduction
- SDK Development Environment
- SDK Project Creation
- GNU Development Tools: GCC, AS, LD, Binutils
- ***Software Settings***
  - *Software Platform Settings*
  - Compiler Settings
- Address Management
- Object File Sections
- Linker Script
- Summary

Minimal Required Services

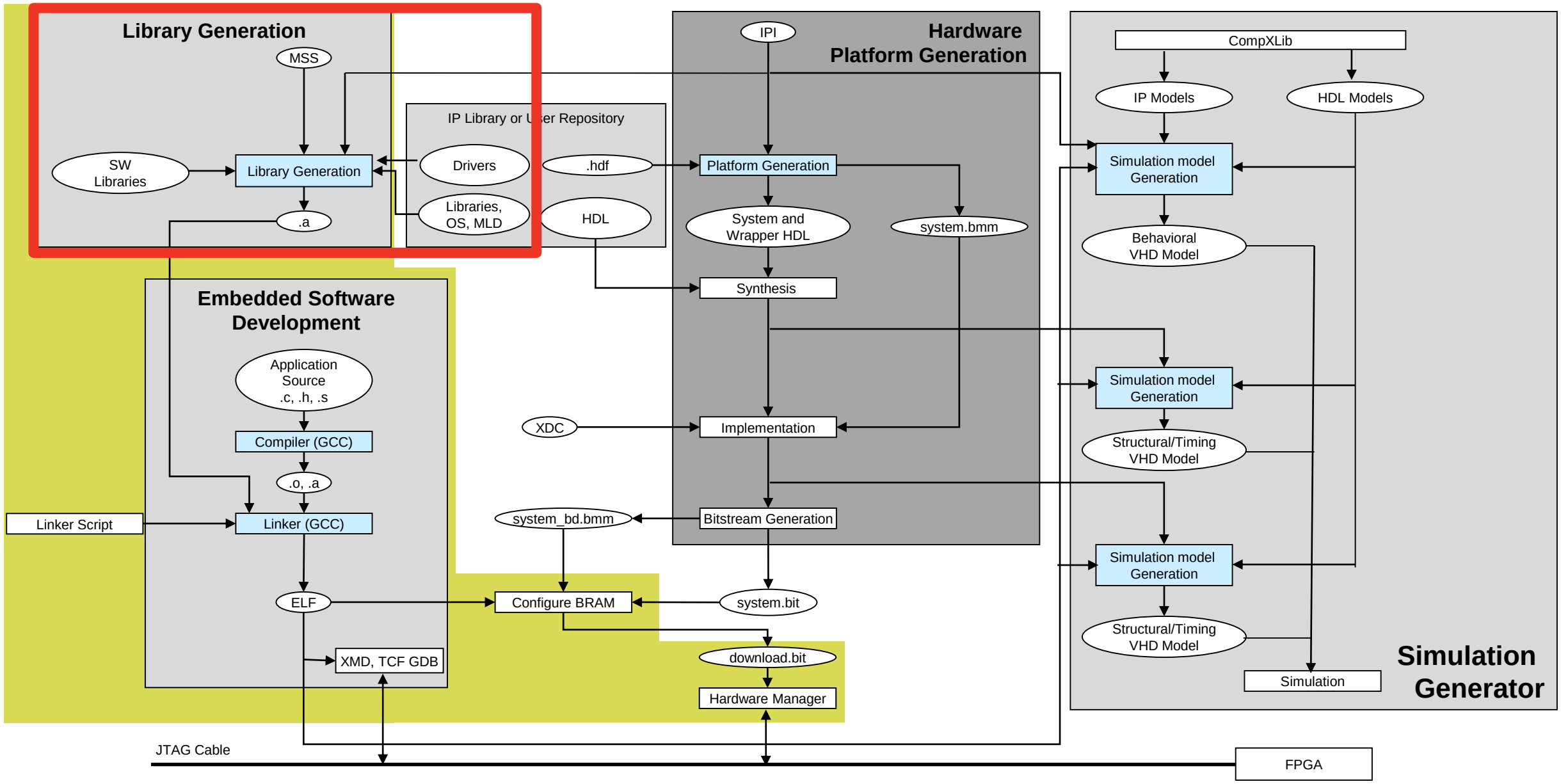
- **C language standard services**
  - C language construct services
  - *stdin* and *stdout*
  - Math library
  - *malloc*
- **Processor support requires these services**
  - Interrupt
  - Cache
  - Language environment support

Accessing Software Platform Properties

- Select the created board support package in the Project Explorer view
- Xilinx Tools > Board Support Package Settings
- Sets all of the software BSP related options in the design
- Has multiple forms selection
  - Overview
  - Standalone
  - Drivers
  - CPU
- As individual Standalone services are selected a configurable menu selection item will appear



Embedded Tool Flow (SDK)



Library Generation Flow (in SDK)

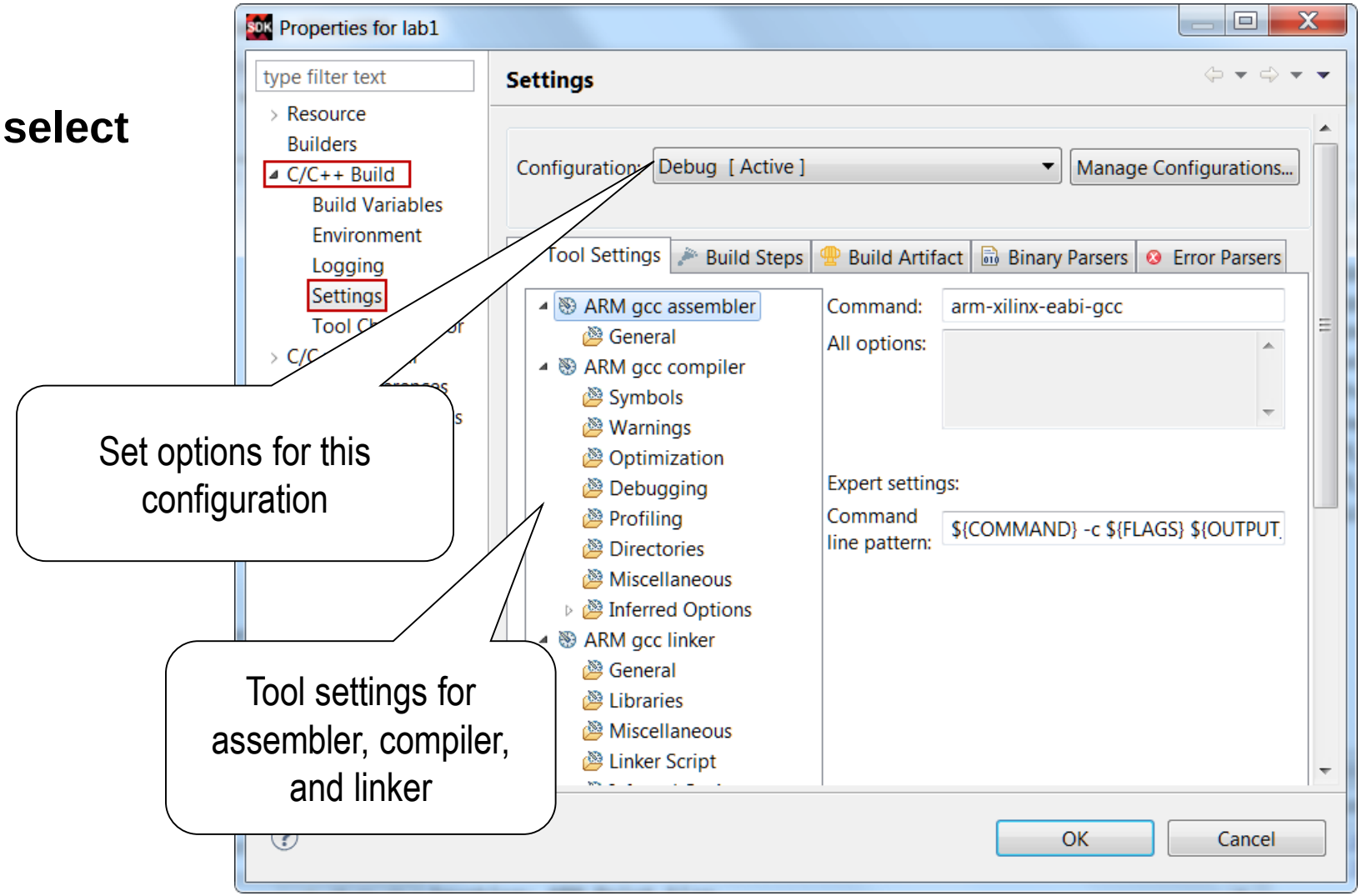
- **Input files → MSS**
  - Output files → libc.a, libXil.a, libm.a
  - Library generator is generally the first tool run to configure libraries and device drivers
    - The MSS file defines the drivers associated with peripherals, standard input/output devices, and other related software features
  - Library generator configures libraries and drivers with this information and produces an archive of object files:
    - libc.a - Standard C library
    - libXil.a - Xilinx library
    - libm.a - Math functions library

Outline

- Introduction
- SDK Development Environment
- SDK Project Creation
- GNU Development Tools: GCC, AS, LD, Binutils
- ***Software Settings***
  - Software Platform Settings
  - *Compiler Settings*
- Address Management
- Object File Sections
- Linker Script
- Summary

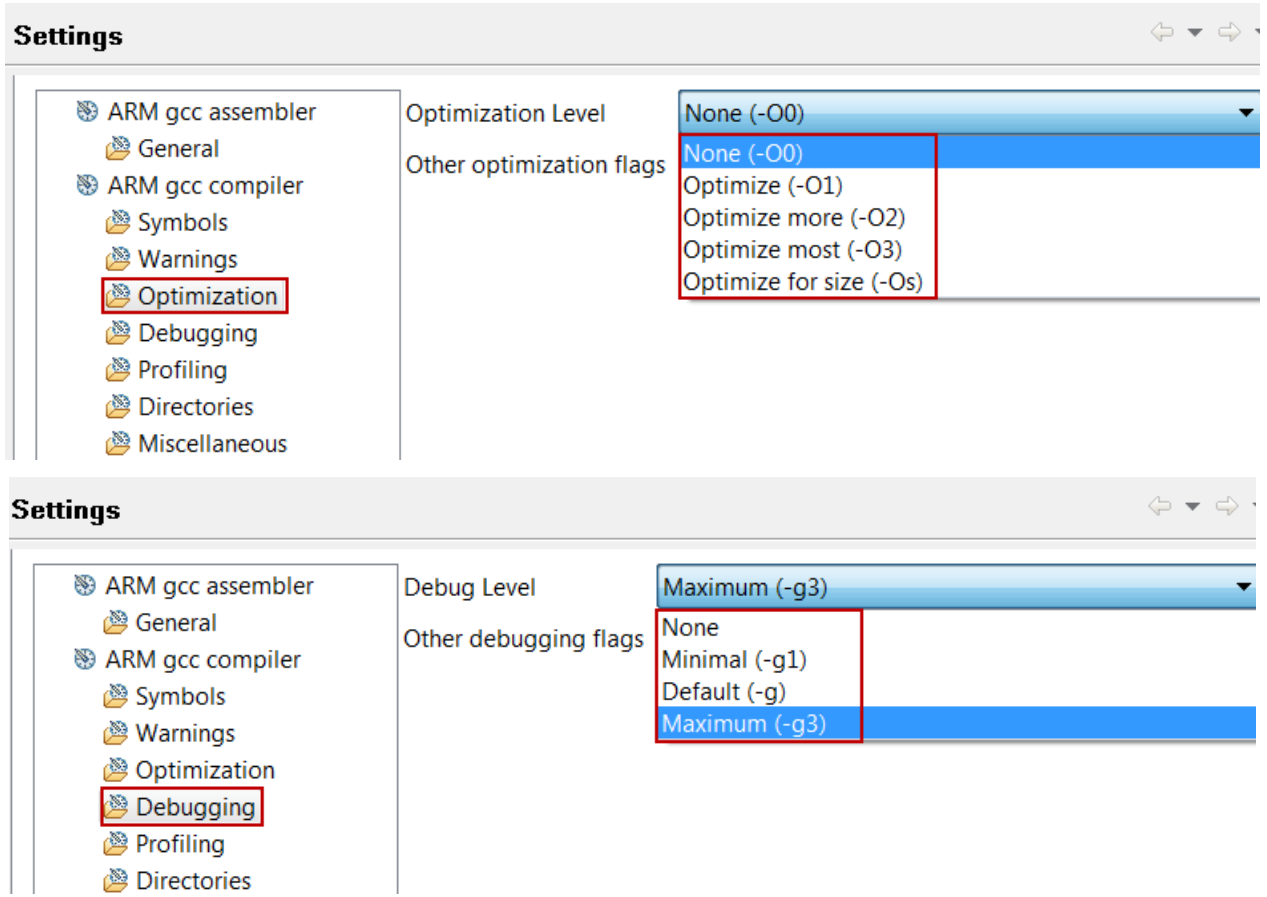
C/C++ Build Settings

- Right-click the top level of an application project and select C/C++ Build Settings
- Most-accessed properties are in the C/C++ Build panel Settings tab
- Each configuration has its own properties



Debug/Optimization Properties

- **Compiler optimization level**
  - None
  - Low
  - Medium
  - High
  - Size Optimized
- **Enable debug symbols in executable**
  - Necessary for debugging
  - Set optimization level to none if possible

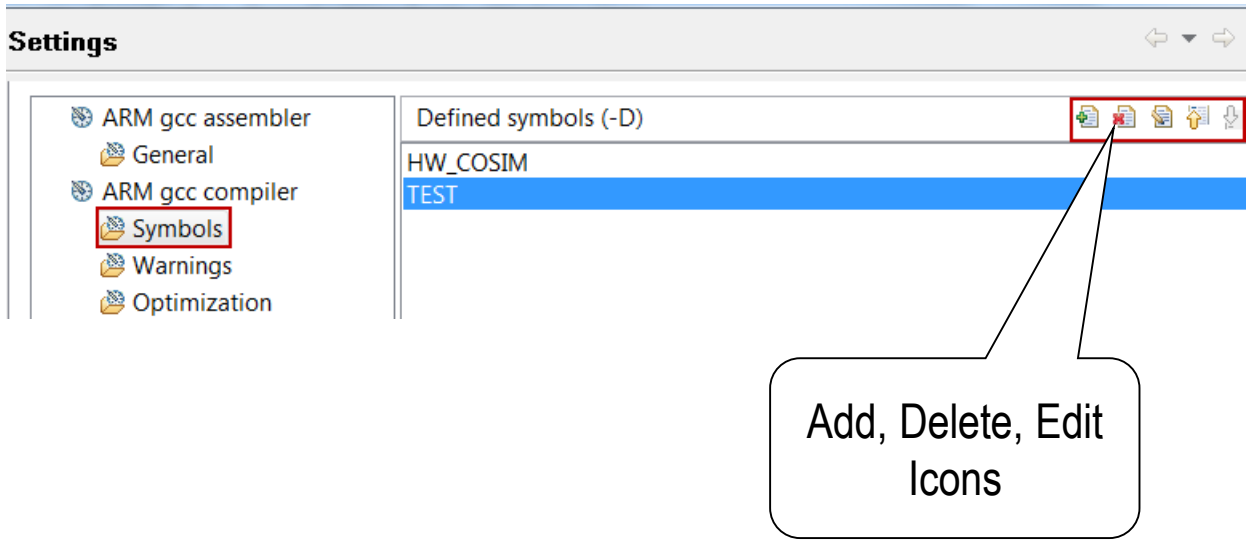


Miscellaneous Compiler Properties

- **Define symbols for conditional compiling**
  - Add
  - Delete
  - Edit
- **References C source**

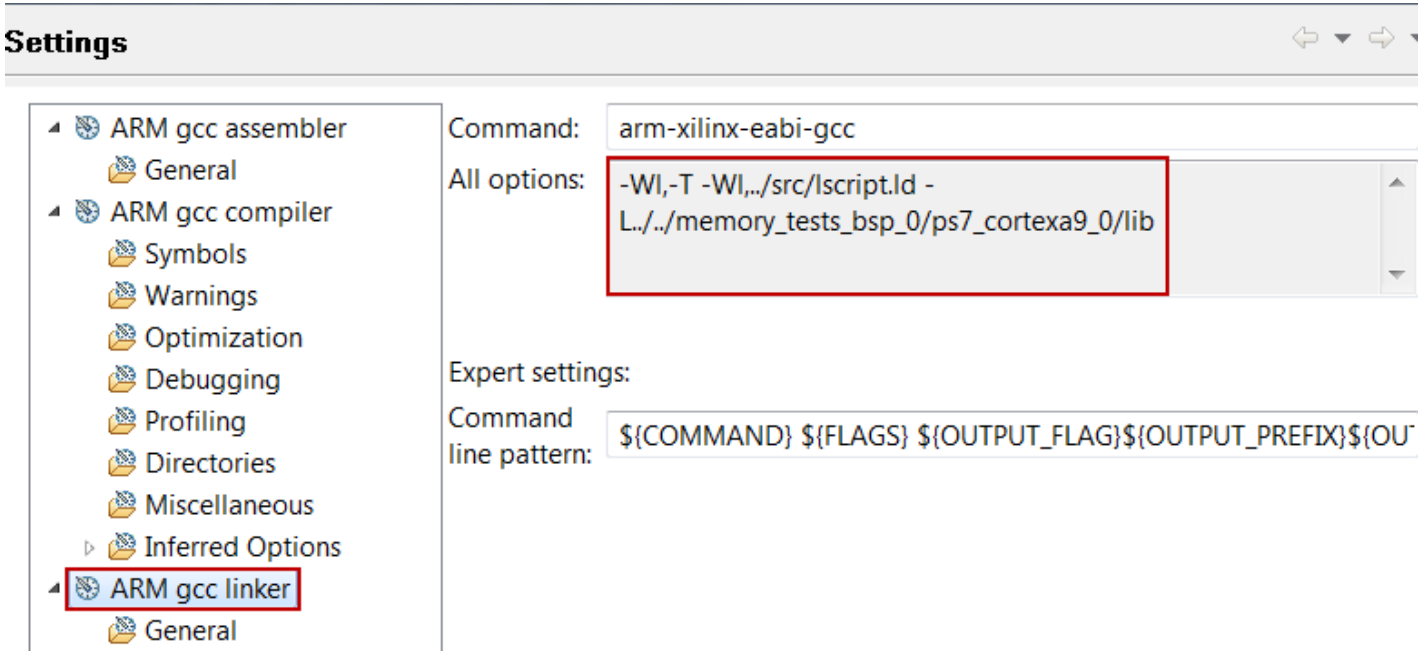
```
#ifdef symbol
conditional statements
#endif
```

- **Passed to compiler as *–D* option**
- **Other compiler options are available**



Linker Properties

- The Root panel displays properties for the selected configuration
- Shown are the linker options for the Debug configuration
- Default settings are fine for simple applications



Outline

- Introduction
- SDK Development Environment
- SDK Project Creation
- GNU Development Tools: GCC, AS, LD, Binutils
- Software Settings
  - Software Platform Settings
  - Compiler Settings
- ***Address Management***
- Object File Sections
- Linker Script
- Summary

Address Management

- **Embedded processor design requires you to manage the following:**
  - Address map for the peripherals
  - Location of the application code in the memory space
    - Block RAM
    - External memory (Flash, DDR3, SRAM)
- **Memory requirements for your programs are based on the following:**
  - The amount of memory required for storing the instructions
  - The amount of memory required for storing the data associated with the program

Standard ARM Programming Model

- **Processing system and programmable logic look the same**
  - AMBA® and AXI interfaces
  - Memory-mapped I/O
  - Register access
- **Consistency for PS and PL = ease of use**
- **Memory map usage: total of 4 GB**
  - 1 GB: DDR RAM
  - 2 GB: dedicated to PL peripherals
  - 1 GB: PS peripherals, OCM, external flash

Start Address	Size	Description
0x0000_0000	1GB	External DDR RAM
0x4000_0000	2GB	Custom Peripherals (Programmable Logic including PCIe)
0xE000_0000	256MB	PS I/O Peripherals
0xF800_0000	32MB	Fixed Internal Peripherals (Timers, Watchdog, DMA, Interconnect)
0xFC00_0000	64MB	Flash Memory
0xFFFC_0000	256KB	On-Chip Memory

Programmer's View of Programmable Logic

- **Programmable logic (PL) memory map**
  - 2 GB total space
    - 1 GB for each AXI master: GP0, and GP1
  - Accessible from any processing system (PS) master
    - Either Cortex-A9 CPU
    - PS DMA engine
    - PS peripheral DMA engine
      - Ethernet
      - USB
      - SD/SDIO

Custom Peripheral

Start Address	Description
0x4000_0000	Accelerator #1 (Video Scaler)
0x6000_0000	Accelerator #2 (Video Object Identification)
0x8000_0000	Peripheral #1 (Display Controller)

Code Snippet

```
int main() {

int *data = 0x1000_0000;
int *accel1 = 0x4000_0000;

// Pure SW processing
Process_data_sw(data);

// HW Accelerator-based processing
Send_data_to_accel(data, accel1);
process_data_hw(accel1);
Recv_data_from_accel(data, accel1);
}
```

Address Map: I/O Peripherals (Zynq AP SoC)

Register Base Address	Description
E000_0000, E000_1000	UART Controllers 0, 1
E000_2000, E000_3000	USB Controllers 0, 1
E000_4000, E000_5000	I2C Controllers 0, 1
E000_6000, E000_7000	SPI Controllers 0, 1
E000_8000, E000_9000	CAN Controllers 0, 1
E000_A000	GPIO Controller
E000_B000, E000_C000	Ethernet Controllers 0, 1
E000_D000	Quad-SPI Controller
E000_E000	Static Memory Controller (SMC)
E010_0000, E010_1000	SDIO Controllers 0, 1
E020_0000	IOP Bus Configuration

Address Map: SLCR Registers (Zynq AP SoC)

Register Base Address	Description
F800_0000	SLCR write protection lock and security
F800_0100	Clock control and status
F800_0200	Reset control and status
F800_0300	APU control
F800_0400	TrustZone control
F800_0500	CoreSight SoC debug control
F800_0600	DDR DRAM controller
F800_0700	MIO pin configuration
F800_0800	MIO parallel access
F800_0900	Miscellaneous control
F800_0A00	On-chip memory (OCM) control
F800_0B00	I/O buffers for MIO pins (GPIOB) and DDR pins (DDRIOB)

Address Map: PS Registers (Zynq AP SoC)

Register Base Address	Description
F800_1000, F800_2000	Triple timer counter 0, 1
F800_3000	DMAC when secure
F800_4000	DMAC when non-secure
F800_5000	System watchdog timer (SWDT)
F800_6000	DDR DRAM controller
F800_7000	Device configuration interface (DevC)
F800_8000	AXI_HP 0 high performance AXI interface w/ FIFO
F800_9000	AXI_HP 1 high performance AXI interface w/ FIFO
F800_A000	AXI_HP 2 high performance AXI interface w/ FIFO
F800_B000	AXI_HP 3 high performance AXI interface w/ FIFO
F800_C000	On-chip memory (OCM)
F800_D000	Reserved
F880_0000	CoreSight debug control

Address Map: CPU Private Bus Registers

Register Base Address	Description
F890_0000 to F89F_FFFF	Top-level interconnect configuration and Global Programmers View (GPV)
F8F0_0000 to F8F0_00FC	SCU control and status
F8F0_0100 to F8F0_01FF	Interrupt controller CPU
F8F0_0200 to F8F0_02FF	Global timer
F8F0_0600 to F8F0_06FF	Private timers and private watchdog timers
F8F0_1000 to F8F0_1FFF	Interrupt controller distributor
F8F0_2000 to F8F0_2FFF	L2-cache controller

Outline

- Introduction
- SDK Development Environment
- SDK Project Creation
- GNU Development Tools: GCC, AS, LD, Binutils
- Software Settings
  - Software Platform Settings
  - Compiler Settings
- Address Management
- ***Object File Sections***
- Linker Script
- Summary

Object File Sections

- **What is an object file?**
  - An object file is an assembled piece of code
    - Machine language:
      li r31,0 = 0x3BE0 0000
  - Constant data
  - There may be references to external objects that are defined elsewhere
  - This file may contain debugging information

Object File Sections
Sectional layout of an object or an executable file

.text	Text section
.rodata	Read-only data section
.sdata2	Small read-only data section (less than eight bytes)
.sbss2	Small read-only uninitialized data section
.data	Read-write data section
.sdata	Small read-write data section
.sbss	Small uninitialized data section
.bss	Uninitialized data section

Sections Example

```
int ram_data[10] = {0,1,2,3,4,5,6,7,8,9};        /* DATA */

const int rom_data[10] = {9,8,7,6,5,4,3,2,1};    /* RODATA */

int I;   /* BSS */

main(){

...
I = I + 10;  /* TEXT */
...

}
```

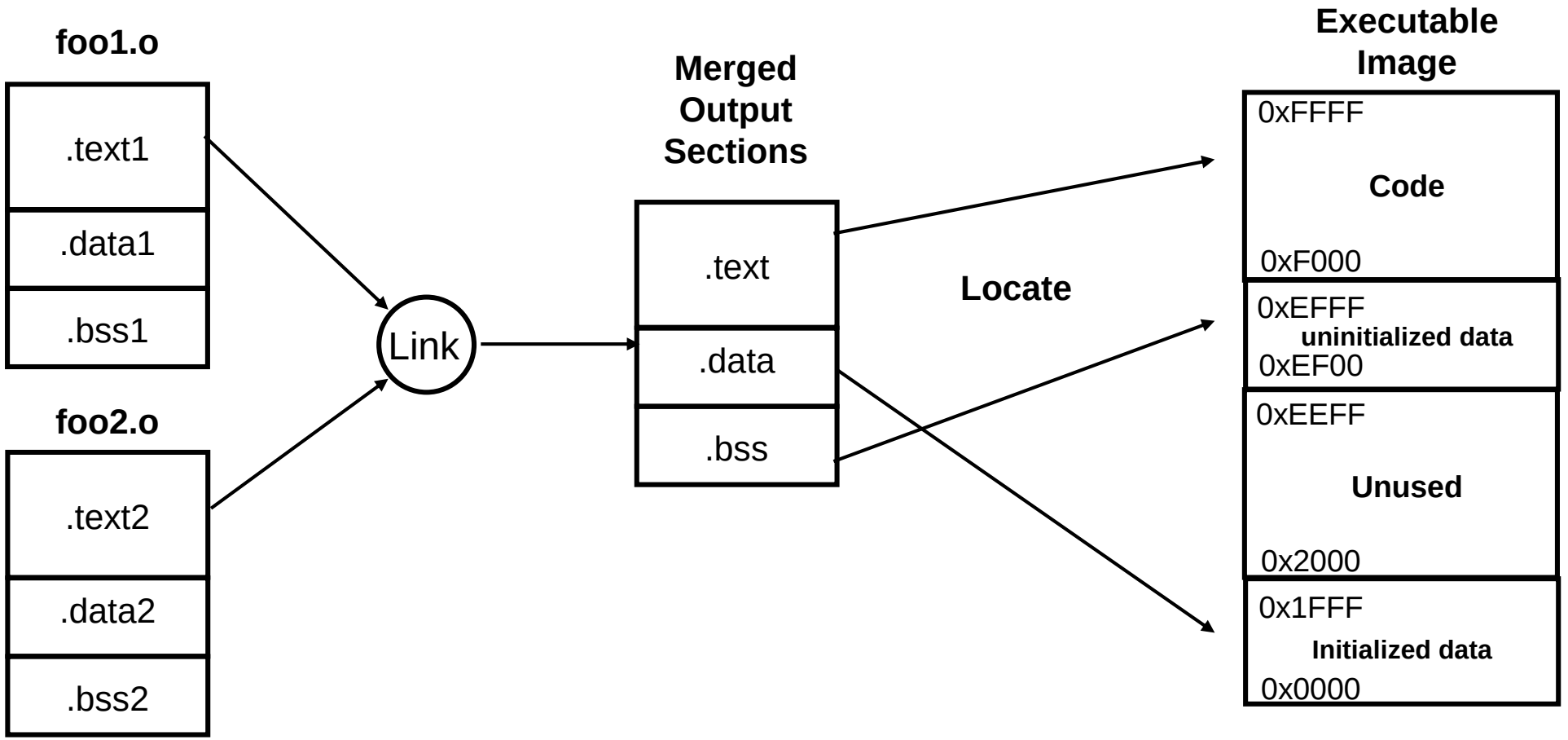
Outline

- Introduction
- SDK Development Environment
- SDK Project Creation
- GNU Development Tools: GCC, AS, LD, Binutils
- Software Settings
  - Software Platform Settings
  - Compiler Settings
- Address Management
- Object File Sections
- ***Linker Script***
- Summary

Linker Script

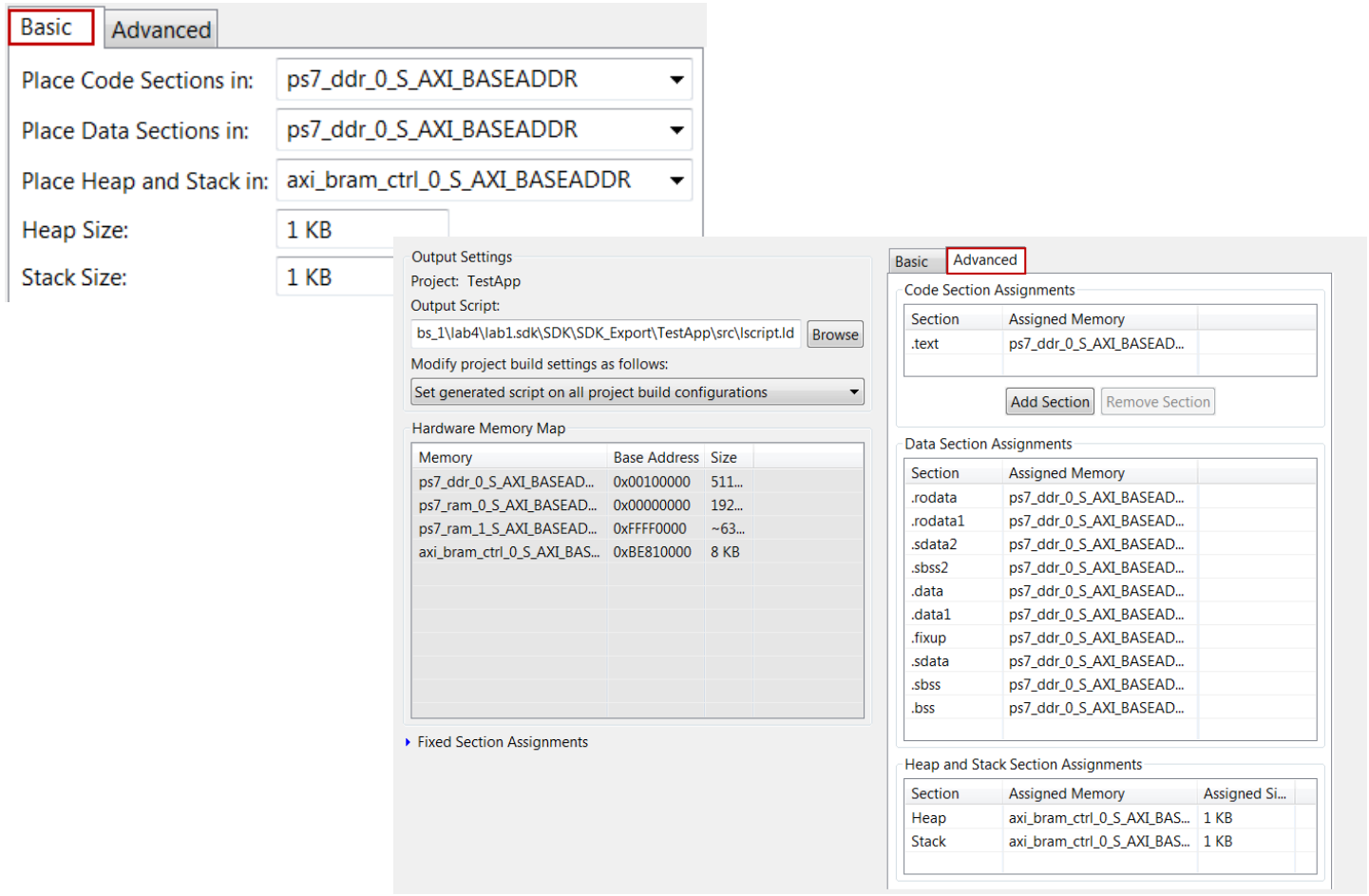
- **Linker script controls the linking process**
  - Map the code and data to a specified memory space
  - Set the entry point to the executable
  - Reserve space for the stack
- **Required if the design contains a discontinuous memory space**

Linker and Locator Flows



Linker Script Generator GUI

- **Table-based GUI allows you to define the memory space for code and data sections**
- **Launch from Xilinx Tools > Generate Linker Script, or from the C/C++ perspective, right-click on <project> > Generate Linker Script**
- **The tool will create a new linker script (the old script is backed up)**



Outline

- Introduction
- SDK Development Environment
- SDK Project Creation
- GNU Development Tools: GCC, AS, LD, Binutils
- Software Settings
  - Software Platform Settings
  - Compiler Settings
- Address Management
- Object File Sections
- Linker Script
- ***Summary***

Summary

- Software development for an embedded system in FPGA imposes unique challenges due to unique hardware platform
- SDK provides many rich perspectives which enable ease of accessing information through related views
- GNU tools are used for compiling C/C++ source files, linking, creating executable output, and debugging
- Software platform settings allow inclusion of software library support
- Compiler settings provide switches including compiling, linking, debugging, and profiling

Summary

- **Embedded processor design requires you to manage**
  - Peripheral address space
  - Memory address space to store data and instructions
    - Internal block memory
    - External memory
- **Linker script is required when the software segments do not reside in a contiguous memory space**