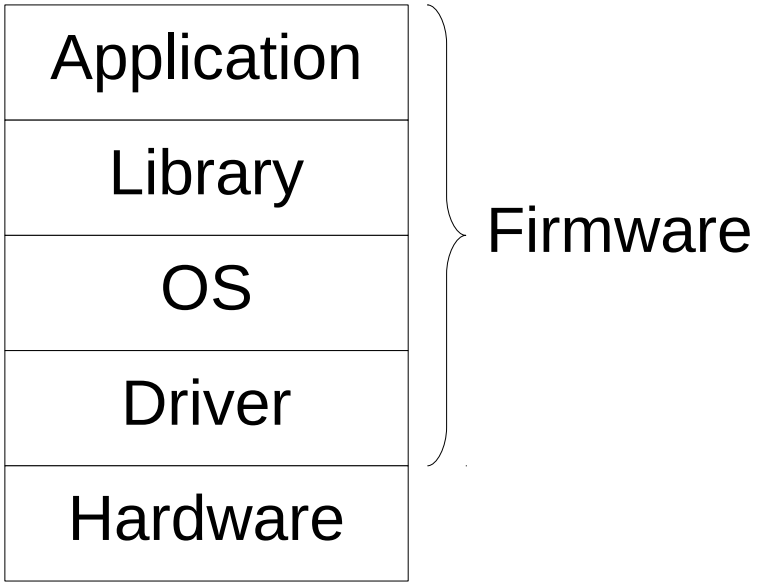
Introduction to Embedded Development

Chapter 1

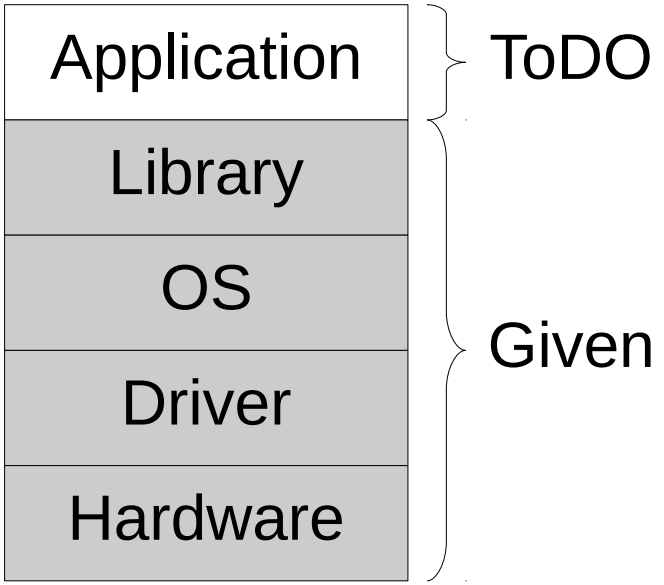
System Stack

- Firmware (run in device)
  - Libraries include standard libraries and system-specific libraries
- CS
  - Application, library, OS
- CE
  - Library, OS, driver
- EE
  - Driver, hardware



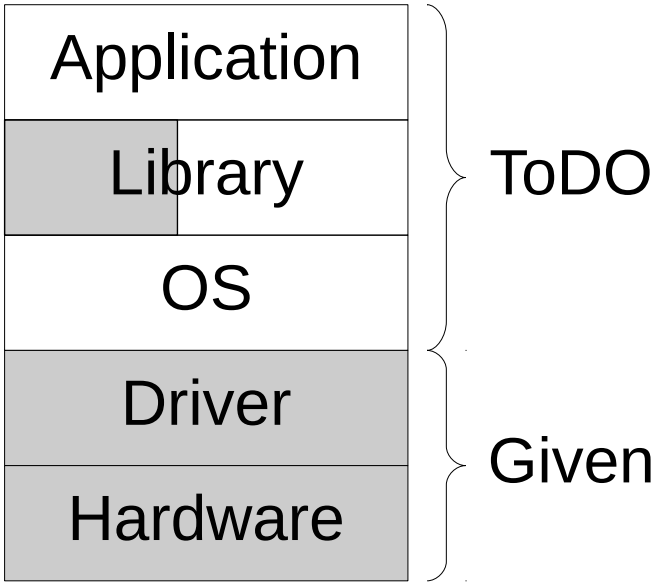
Developmental Systems

- Application development
  - Development on top of existing drivers and OS with given APIs
  - No direct programming interface to hardware
  - Develop with a well established software development kits (SDK)
  - Android, iPhone
  - Arduino, Raspberry Pi, Busybox, Beaglebone



Developmental Systems

- System development (for this class)
  - Need to develop OS, system-specific libraries, and applications
  - Starter kits: PIC24E, STK500
  - Customized board
  - Develop on bare metal

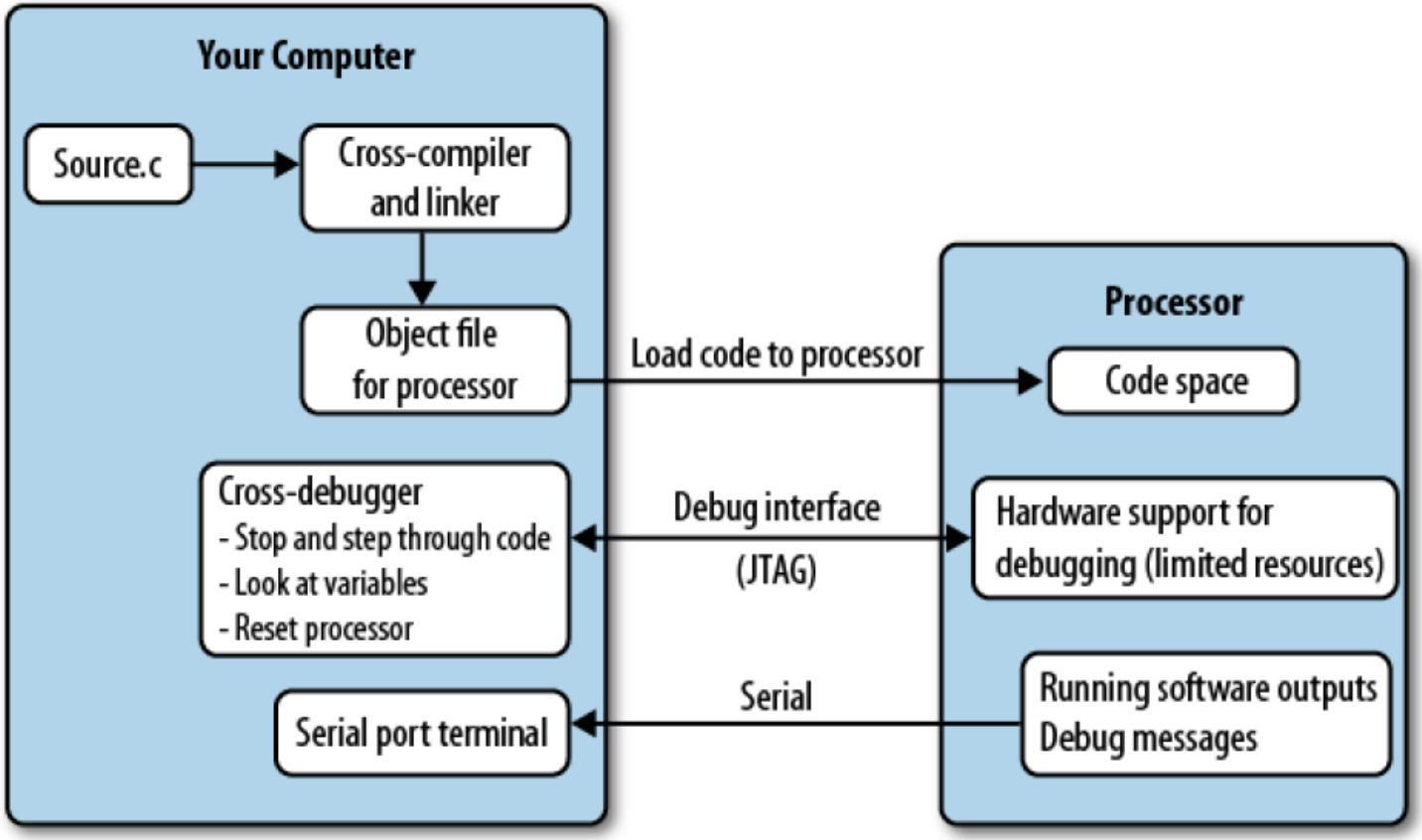


Embedded Developmental System

- Not embedded system, but for development
- Software
  - Code editor (IDE/Studio)
  - Compiler (linker)
  - Standard libraries
  - Chip-specific drivers
- Hardware
  - Development board (chip and power)
  - Programmer or flasher
  - Debugger
  - Extended peripheral modules

Software Development

- For system development only
- Cross-compiler, cross-debugger



Programming Language

- No single language perfect for every system
  - Designed to perform a dedicated function
- Machine code
  - atmega128, pic24ep512ug810
  - Binary, hardware dependent
- Assembly
  - AVR: atmega128
  - PIC: pic24ep512ug810
- High level
  - System development: C
  - Application development: C, Java, Swift (Object-C)

Code Editor

- IDE or Studio
  - Microchip Mplab X
    - http://www.microchip.com/development-tools/downloads-archive
  - Atmel Studio
    - http://www.atmel.com/tools/studioarchive.aspx
  - Project configuration
    - Compiler
    - Libraries
      - Include
      - Link
    - Programmer
    - Debugger

Practice (Case 2)

- New project
  - Project type: standalone
  - Chip family
  - Development windows
- Project property
  - Source folder
  - Chip
  - Programmer/debugger
    - Compilation
    - Linking

Compiler

- Source code -> executable
  - Preprocess -> source code
  - Compile -> object code
  - Link -> executable code
- Host : where a program is compiled
- Target : where a program is executed
- Host = Target in regular computer
- In embedded system
  - Host is a regular computer
  - Target is the embedded system
  - Cross-compilation : avr-gcc or xc16-gcc

Cross-compilation

- Source to executable
  - Coding or pre-processing at host
    - C
  - Compile at host
    - C -> machine code
    - Source file -> object file
  - Link at host
    - Add library and system codes
    - Object file -> executable file
  - Extract at host
    - Executable -> target machine code (native code)
  - Upload and execute : target

GCC based X-Compiler

- Command line: `xxx-gcc sourcefiles options`
  - **`-o`** `outfile` : file to generate
  - **`-c`** : compile only
  - **`-Wall`** : show all warnings
  - **`-g`** : include debugging information
  - **`-O`**`[0-3s]` : optimization
  - **`-mmcu=`**`tgt` or **`-mcpu=`**`tgt` : target processor
  - **`-D`**`macro` : additional macro
  - **`-I`**`dir` : additional include directory
  - **`-L`**`dir` : additional library directory
  - **`-l`**`lib` : additional library
  - **`-Wl,`**`[lflags]` : linking flags

Cross Compiler (Case 1)

- hello.x86.asm
  - .text section
    - <_start>
    - call <_libc..main@plt>
    - <main>
    - call <puts@plt>
    - ret
    - <puts@plt>
    -
- hello.avr.asm
  - .text section
    - <_vectors>
    - jmp <__ctors_end>
    - <main>
    - call <puts>
    - jump <_exit>
    - <puts>
    - <_exit>

Cross Compiler

- Sections: structures in executable
  - Corresponding to memory layouts of MCUs
  - Support by compilers
  - Specified by MCUs' architectures
- Tags:
  - Section tags: .text, .data, .plt, .got
  - Block tags: <main>, <_vectors>
  - Specified by compilers

Cross Compiler

- Assembly languages
  - Not machine code
  - x86: specified by Intel
  - avr: specified by Atmel
  - pic: specified by Microchip
- Executable formats
  - Representation of machine code
  - In host
    - ELF (Executable and Linkable Format)
      - System V Application Binary Interface specification
      - Tool Interface Standard
    - COFF (Common Object File Format)
  - In target
    - IHex (Intel Hex): by Intel

Cross Compiler

- For building native programs
- Open source based
  - C cross-compiler: avr-gcc
    - Compiled to ELF in host
  - Code extractor: avr-objcopy
    - Extracted to HEX for card
  - Code loader: uisp
    - Load code to card
- Proprietary
  - xc16-gcc
    - http://www.microchip.com/mplab/compilers
  - xc16-bin2hex

Libraries (PIC)

- Standard libraries
  - Come with compiler, standard C
  - MPLAB® XC16 User Guide
    - http://ww1.microchip.com/downloads/en/DeviceDoc/50002071F.pdf
  - 16-Bit Language Tools Libraries Reference Manual
    - http://ww1.microchip.com/downloads/en/DeviceDoc/50001456J.pdf
- Support libraries
  - Need to download and install separately
  - Peripheral libraries (PIC24)
    - http://www.microchip.com/mplab/compilers
  - Application libraries
    - http://www.microchip.com/mplab/microchip-libraries-for-applications
  - Example codes

Libraries (AVR)

- Standard library
  - http://www.nongnu.org/avr-libc/
- Support libraries
  - Device drivers
  - Example codes
  - http://www.atmel.com/devices/ATMEGA128.aspx?tab=documents

Debug with Support

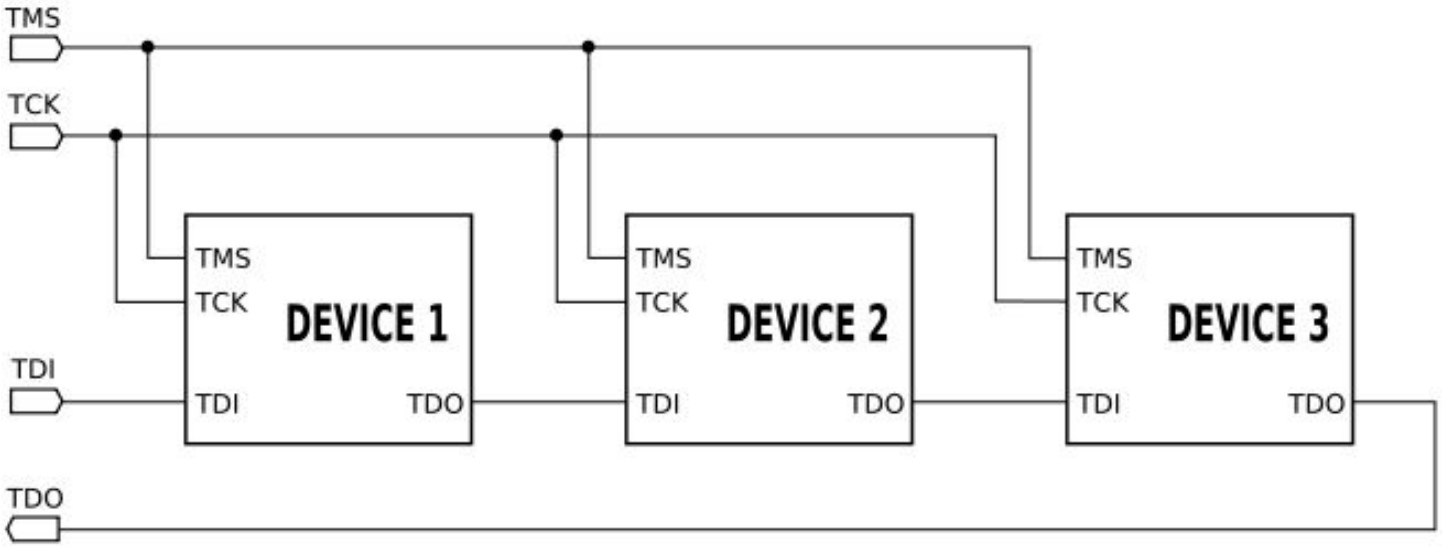
- Resources in processor to support debugging.
  - Breakpoints
  - Execution control: run, pause, resume
  - Watch: register, memory, instruction
- Debug interface between host and processor.
  - JTAG
- Emulator : debugging device
  - In-circuit emulator (ICE)
  - JTAG adapter

JTAG

- Debug support is for many software developers the main reason to be interested in JTAG.
- Debugging architectures built up using JTAG, such as ARM CoreSight, Nexus, ...
- Processors can normally be halted, single stepped, or let run freely.
- Code breakpoints are supported, both for code in RAM and in ROM/flash.
- Data breakpoints are often available, as is bulk data download to RAM.

JTAG

- JTAG pins
  - TDI (Test Data In) : input JTAG instructions
  - TDO (Test Data Out) : obtain scanned data
  - TCK (Test Clock) : clock, SPI
  - TMS (Test Mode Select) : state control
  - TRST (Test Reset) optional



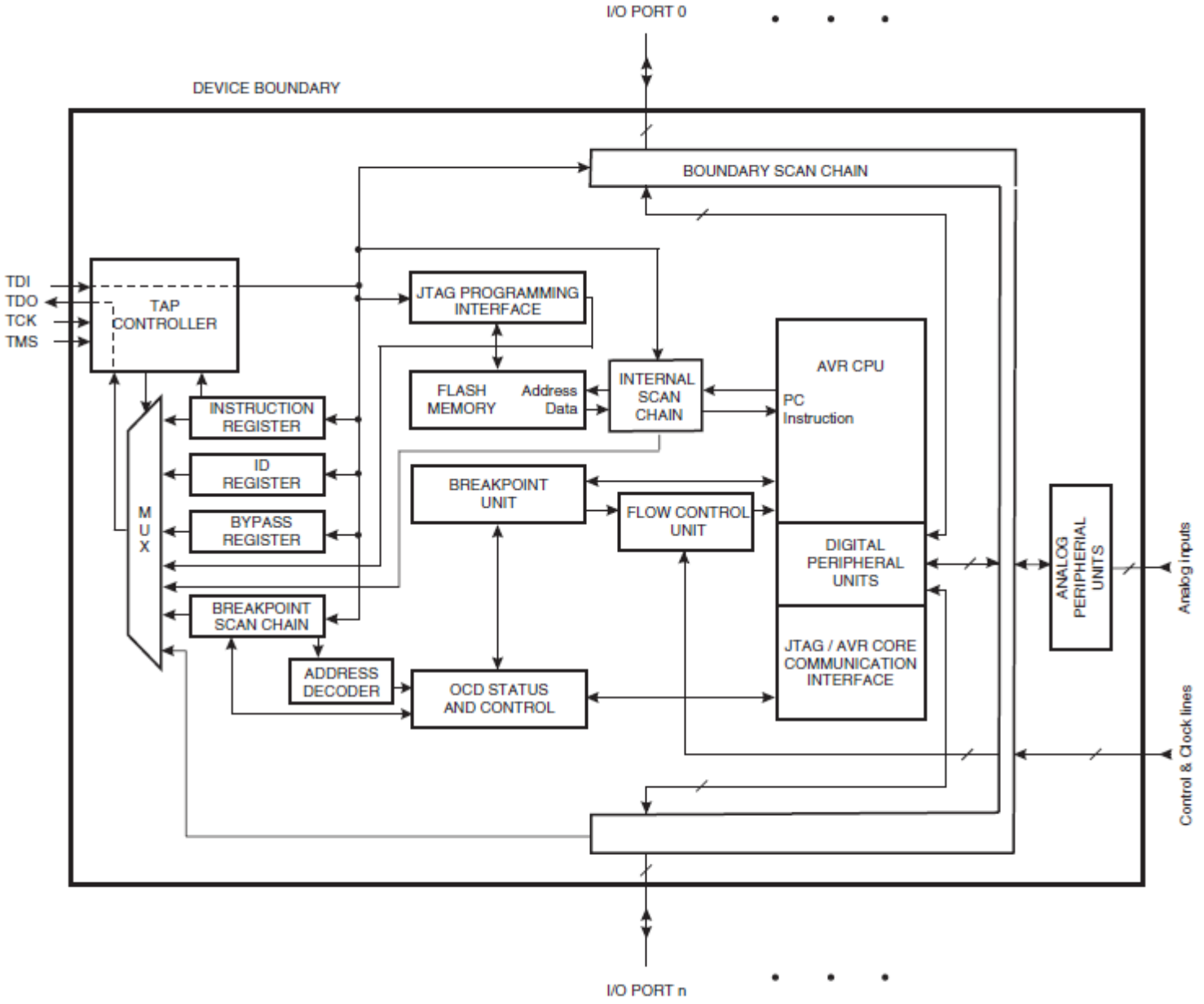
JTAG

- Examples of JTAG instructions
  - EXTEST : for external testing, such as using pins to probe board-level behaviors
  - INTEST : for internal testing, such as using pins to probe on-chip behaviors
  - PRELOAD : loading pin output values before EXTEST
  - SAMPLE : reading pin values into the boundary scan register
  - HIGHZ : deactivates the outputs of all pins
  - RUNBIST : places the chip in a self-test mode
  - SCAN_N : configures a scan path select register (SCREG) affecting the signals to which other boundary scan operations apply
  - USERCODE : returns a user-defined code, for example to identify which FPGA image is active
  - HALT, RESTART : halts or restarts the CPU.

JTAG Example

- Boundary-scan Capabilities
- Debugger Access to:
  - All Internal Peripheral Units
  - Internal and External RAM
  - The Internal Register File
  - Program Counter
  - EEPROM and Flash Memories
- Extensive On-chip Debug Support for Break
  - AVR Break Instruction
  - Break on Change of Program Memory Flow
  - Single Step Break
  - 4 : Program Memory Breakpoints + Data Memory Breakpoints
- Programming of Flash, EEPROM, Fuses, and Lock Bits
- On-chip Debugging Supported by AVR Studio

JTAG Example



Debug without Support

- Printf
  - Communication port between host and processor
  - Modified stdio library
- Through peripherals
  - LEDs
- Debugging is always a headache, but **extremely important and necessary.**
  - Always change timing of code execution
  - Limited debugging capability

Debug with Simulator

- **Before you debug on board, debug in simulator.**
- What a simulator can do
  - Control execution
  - Watch all registers and memory
  - Simulate input, output and interrupts
  - Accurate timing
- What a simulator cannot do
  - Get input from real

Practice (Debug, Case 2)

- Debug toolbar
- Breakpoint window
- Variable window
  - Local
  - Watch
- Processor window
  - Registers
  - Memory
- Disassembly window

- Breakpoint
  - Continue
  - Stop
- Step
  - Over
  - Into
  - Instruction
- Watch
  - Variable
  - Registers
  - Memory

Debug

- General debug flow
  - Get a **running problem** (not compilation)
  - Choose the target code block that **may** cause the problem
  - Set break points **before** the target code
  - Run to the break points
  - Check all variables are correct **as expected** at the break points (!important!)
  - Step over line by line
    - Check all variables are correct **as expected** at each line
  - Reach the end of target block
    - Either find the erroneous line of code
    - Or choose another target code block to debug