BECE403E-EMBEDDED SYSTEM DESIGN

MODULE-3

EMBEDDED SOFTWARE DEVELOPMENT ENVIRONMENT

MODULE-3

Embedded Software Development Environment

Cross assemblers/compilers, Linker, Runtime Library, Pre-processor Workflow, make files, Compiler Tool chains – gcc & ARM, Device Driver, Firmware, Middleware - Debugging tools: Emulators, Simulators, In-Circuit Debuggers, Logic Analyzer, Integrated Development Environment (IDE).

EMBEDDED SYSTEM BUILD PROCESS

EMBEDDED SYSTEM BUILD PROCESS

EMBEDDED SOFTWARE PROGRAMMING LANGUAGES

- Developers use a variety of programming languages in embedded systems which includes C, C++, Python, MicroPython, and Java.

- Among many programming languages, C-Programming has benefits for both low level hardware interactions and high level software language features.

- Typical embedded engineers write a form of C program called Embedded C which focus on the following features.
  - ✓ Processor independent
  - ✓ Efficient memory management
  - ✓ Timing centric operations
  - ✓ Direct hardware/IO control
  - ✓ Code size constraints
  - ✓ Optimized execution
  - ✓ Portability

EMBEDDED SYSTEM BUILD PROCESS

EMBEDDED SOFTWARE PROGRAMMING LANGUAGES

- Build Process is the process of converting the high level source code(Ex. C) representation of your embedded software into an executable binary image.

- This Process involves many steps but the main three distinct steps of this process are:
  1. Object file: Each of the source files must be compiled or assembled into an object file.
  2. Relocatable program: All of the object files that result from the first step must be linked together to produce a single object file, called the relocatable program.
  3. Relocation: Physical memory addresses must be assigned to the relative offsets within the relocatable program in a process called relocation.

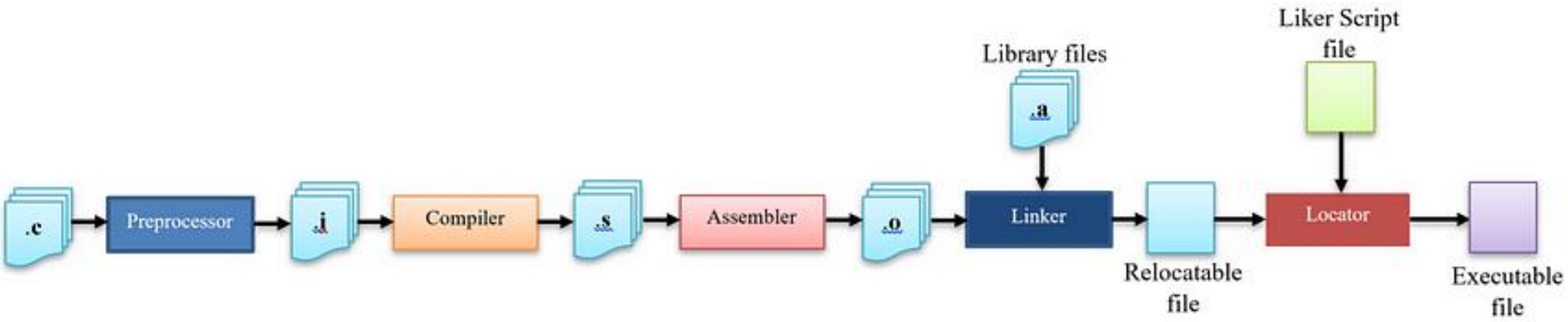
- List of tools required for Build process are: Cross assemblers/compilers, Runtime Library, Linker, Locator, Debugger

EMBEDDED SYSTEM BUILD PROCESS

EMBEDDED SOFTWARE PROGRAMMING LANGUAGES

- To develop software for a General Purpose Computer
  - Create source file
  - Type in C code
  - Build: compile and link
  - Execute: load and run
- To develop software for an embedded system
  - Create source file
  - Compiler: translate into assembly program
  - Assembler: translates the assembly program into machine code i.e object file
  - Linker: combine all object files and libraries, resolve all symbols
  - Locater: assign memory addresses to code and data
  - Download: copy executable image into Target processor memory
  - Execute: reset Target processor

EMBEDDED SYSTEM BUILD PROCESS



EMBEDDED SYSTEM BUILD PROCESS

PREPROCESSOR

- Pre-processing is not a part of the compiler, but is a separate step in the compilation process in which all the pre-processor directives are evaluated.
- Pre-processing is a process of running a set of instructions provided by the programmer explicitly as code before the program compiles.
- The pre-processor provides the ability for the inclusion of header files, macro expansions, conditional compilation, and line control.
- Pre-processor instructions are called pre-processor directives, and they all start with a hash symbol (#). Few examples:
  - Inclusion of source files Ex: #include<stdio.h>, #include "file“
  - Macro expansion Ex: #define MIN 234
  - Conditional compilation Ex: #ifdef DEBUG printf("debugging...") #endif

EMBEDDED SYSTEM BUILD PROCESS

COMPILER

- Compiler translate programs written in high-level programming language (C or C++) to a low-level language (assembly code) for a particular processor.
- This conversion is not a one to one mapping of lines but instead a decomposition of C operations into numerous assembly operations.
- Each processor has its own unique machine language, so you need to choose a compiler that produces programs for your specific target processor.
- Types of Compiler:
  - Native-compiler: Runs on a computer platform (x86) and produces code for same computer platform (x86)
  - **Cross-Compiler:** Runs on one computer platform (x86) and produces code for different architecture (ARM)
- The use of a cross-compiler is one of the defining features of embedded software development.

EMBEDDED SYSTEM BUILD PROCESS

COMPILER

- A compiler operates in logically interrelated phases whereby source code is translated assembly file. List of phases includes,
  - Lexical Analysis: High level program is converted into a sequence of tokens. A token may be a keyword (while'), an operator (*), an identifier (variable name), literal (10 or "my string"), comment.
  - Syntax Analysis: Ensures that tokens are organised in the correct way, according to the rules of the language otherwise compiler will produce a syntax error.
  - Semantic analysis: Checks on the logical structure of the program to detect potential problems such as unused variables, uninitialized variables, etc.
  - Intermediate code generation: Intermediate representation of the final machine code is produced.
  - Code optimization: Code is optimized in order to run faster and efficiently by performing inline expansion of functions, dead code removal, loop unrolling, register allocation, etc.
  - Code generation: The optimized intermediate code is translated into assembly language for a specified machine architecture.
  - Error handling: Errors are detect and reported to enable a programmer to debug a source program.

EMBEDDED SYSTEM BUILD PROCESS

COMPILER – Memory Allocation

- After the code generation is finished, the compiler allocates memory for code and data in sections; each section has different information.

- Different memory segments are,
  - .code / .text - opcodes, Literals
  - .data - initialized global variables (and their initial values),
  - .bss - uninitialized global variables into this section.
  - . const / .rodata - constants
  - .stack / R# - local/Automatic variables (variables are defined within functions)
  - .heap - dynamic objects

- Stack and Heap segments are not allocated by the compiler but by the Linker at link-time.

EMBEDDED SYSTEM BUILD PROCESS

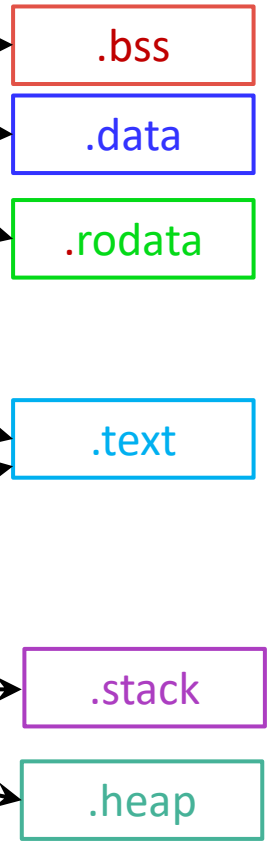
COMPILER – Memory Allocation Example

```
int iVar1;
int iVar2 = 10;
const double dVar1 = 1.0;


void aFunc (int p)
{
  // Function def'n.
}

int main(void)
{
  double loc;
  int *p = malloc(sizeof(int));

  free p;
}
```



EMBEDDED SYSTEM BUILD PROCESS

ASSEMBLER

- Assembler converts the assembly language code into an object file which is an binary file that contains set of machine-language instructions (opcodes) and data sections.
- In short, the assembler generates a microcontroller architecture-specific machine code without absolute addresses.
- Each processor has its own unique machine language, so you need to choose a appropriate assembler that produces programs for your specific target processor.
- **Cross-Assembler** is a type of assembler that is designed to generate machine code for a different processor from the one on which the assembler itself runs.
- Note that compilers nowadays can generate an object code without the need of an independent assembler.

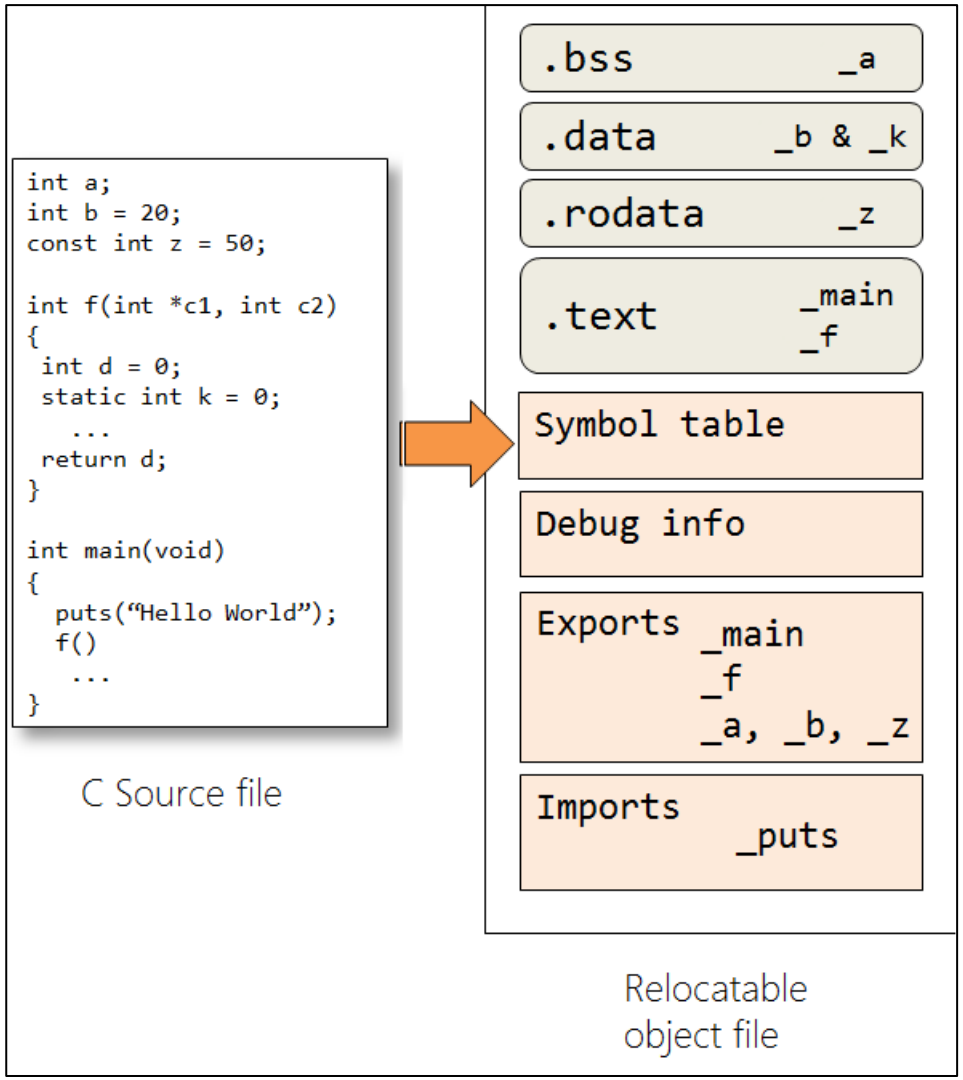
EMBEDDED SYSTEM BUILD PROCESS

ASSEMBLER

- Your embedded application may have reference to built-in libraries and more than one source code file therefore, the assembler may generate multiple object files.
- Note that the object file only contains the sections for static variables. At this stage, section locations are not fixed.
- Each object file contains following sections,
  - Symbol table - to store all variable names and their attributes.
  - Debug info section - mapping between the source code and the information needed by the debugger.
  - Exports section - contains global symbols either functions or variables.
  - Imports section - contains symbol names that are needed from other object files.
- Exports, imports, and symbol table sections are used by the linker during linking stage.

EMBEDDED SYSTEM BUILD PROCESS

ASSEMBLER



EMBEDDED SYSTEM BUILD PROCESS

LINKER

- The role of the linker is to combine multiple object files into a single relocatable file and resolve references between these multiple object files using symbols and references table.

- While combining the object files together, the linker performs the following operations:

1. Symbol resolution:
   - In multi-file program, if there are any references to symbol defined in another file, the assembler marks these references as "unresolved".
   - The linker determines the values of these references from other object files or all specified library/archive (.a) files, and patches the code with the correct values.
   - If, the Linker still cannot resolve a symbol it will report an 'unresolved reference' error.
   - If the linker finds same symbol defined in two object files, it will report a "redefinition" error.

EMBEDDED SYSTEM BUILD PROCESS

LINKER

2. Relocation: Process of changing addresses already assigned to symbols. This will also involve patching up all references to reflect the newly assigned address. Relocation process involves following,

   a) Section Merging: The Linker then concatenates like-named sections from all the input object files. The combined sections (output sections) are usually given the same names as their input sections. Program addresses are adjusted to take account of the concatenation.

   b) Section placement: When a program is assembled each section is assumed to start from address 0. And thus, symbols are assigned values relative to the start of the section. When the final executable is created, the section is placed at some address X. And all references to the symbols defined within the section, are incremented by X, so that they point to the new location.

   c) Data initialisation: Create a shadow data section (.sdata) to copy read-only sections (.code) to RAM to speed up execution

- After merging all of the code and data sections and resolving all of the symbol references, the linker produces a special "relocatable" copy of the program except no memory addresses have yet been assigned to the code and data sections within.

EMBEDDED SYSTEM BUILD PROCESS

LOCATOR

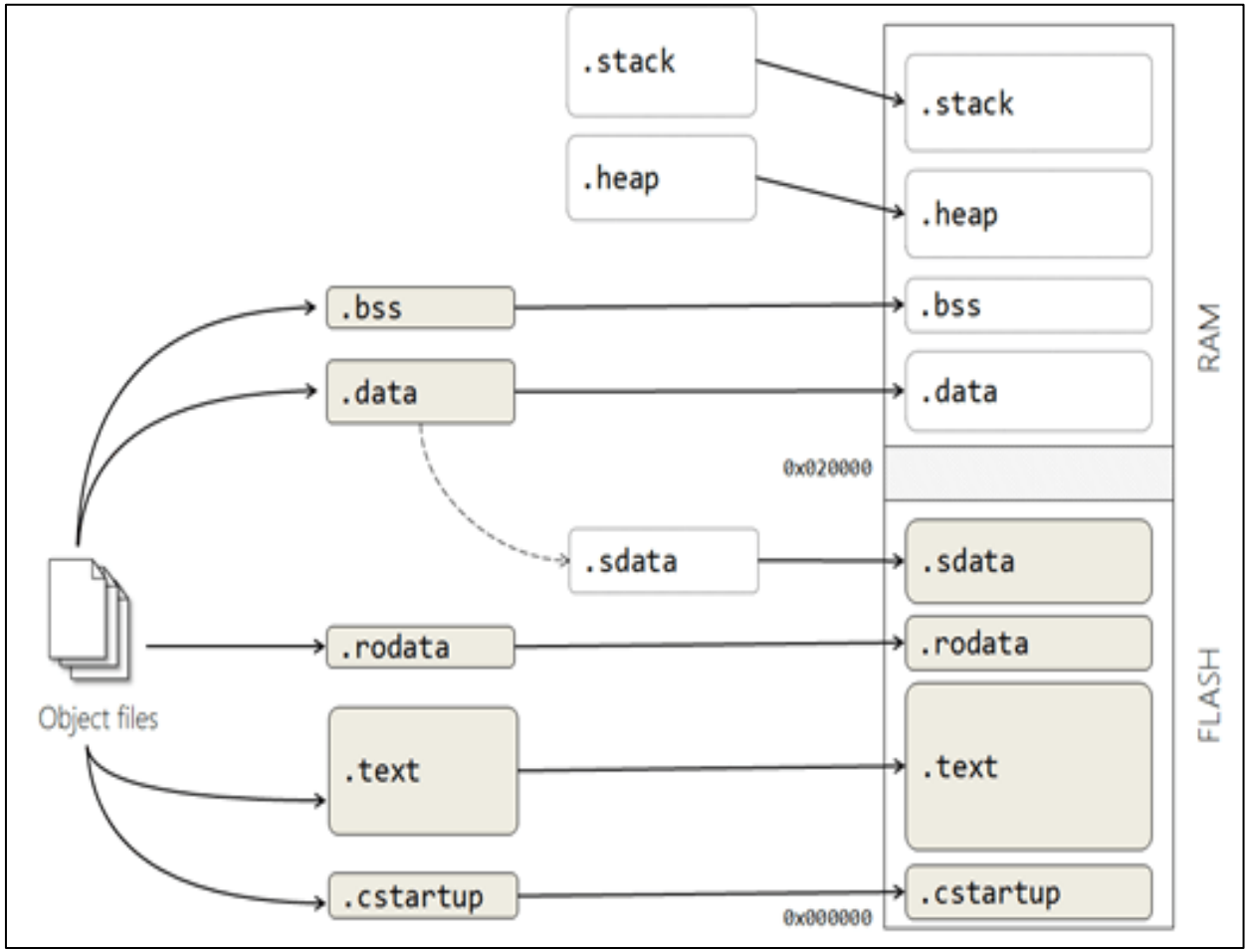
- A locator performs the conversion from relocatable program to executable binary image.
- The locator gets the output file of the linker and maps the code and data into microcontroller memory by assigning physical addresses to code and data.
- The locator will perform checks to ensure that your code and data sections will fit into the designated regions of memory.
- Linker Control File (LCF) also another name for this file is linker-script file, linker configuration file is responsible to inform the locator how to map the executable into proper addresses.
- LCF defines physical memory layout (Flash/SRAM) of the target board and placement of the different program regions to produce a single executable binary file.

EMBEDDED SYSTEM BUILD PROCESS

LOCATOR

- The intermediate output from the locating process is a load file in a platform-independent format, commonly .ELF or .DWARF (although there are many others).
- The ELF file is also used by the debugger when performing source-code debugging.
- In order to be loaded onto the target the ELF file must be converted into a native flash / PROM format (typically, .bin or .hex)
- The output of this final step of the build process is a binary image containing physical addresses for the specific embedded system.
- This executable binary image can be downloaded to the embedded system or programmed into a memory chip.

EMBEDDED SYSTEM BUILD PROCESS



Mapping different sections into microcontroller memory

- .cstartup – the system boot code – is explicitly located at the start of Flash.
- .text and .rodata are located in Flash, since they need to be persistent
- .stack and .heap are located in RAM.
- .bss is located in RAM in this case but is (probably) empty at this point. It will be initialised to zero at start-up.
- .data section is located in RAM (for run-time) but its initialisation section, .sdata, is in ROM.

EMBEDDED SYSTEM BUILD PROCESS

RUN-TIME LIBRARY

- A runtime library is a collection of software routines and functions that provide low-level support for a programming language during the execution of a program.
- It consists of precompiled code that can be linked to a program at either compile time or runtime.
- When the library files remain out of compilation of the main executable and comes into use only when required at runtime, it is called dynamic linking.
- These library files can be custom programmed or provided by the language compiler for standard functions. e.g. C compilers provide C-Standard runtime libraries
- For this reason, some programming bugs are not discovered until the program is tested in a "live" environment – Runtime error

EMBEDDED SYSTEM BUILD PROCESS

MAKEFILE

- Manually entering individual compiler and linker commands on the command line becomes tiresome very quickly. In order to avoid this, a makefile can be used.
- A Makefile is a script file used in software development projects to automate the building and compilation of source code into executable programs or libraries.
- Makefiles are associated with the "make" build automation tool, which reads the instructions from the Makefile and executes the necessary commands to build the project.
- The make utility follows the rules in the makefile in order to automatically generate output files from a set of input source files.
- Makefiles are commonly used in Unix and Unix-like systems, but they can also be used on other platforms with tools like GNU Make.

COMPILER TOOLCHAIN

COMPILER TOOLCHAIN

GCC (GNU Compiler Collection)

- In embedded systems development, the compiler and toolchain are fundamental components used to translate high-level programming code into machine code that can run on a specific microcontroller or processor.
- A toolchain normally consists of a compiler, a linker, and run-time libraries.
- GNU Compiler Collection (GCC) is an open-source compiler suite widely used in the software development community and it can be configured as native or cross-compilers.
- GCC is a collection of compilers for various programming languages, including C, C++, Ada, Fortran, and others.
- The gcc compiler will run on all common PC and supports an impressive set of target processor including AVR, Intel x86, MIPS, PowerPC, ARM, and SPARC.

COMPILER TOOLCHAIN

GCC (GNU Compiler Collection) TOOLCHAIN

- GCC has extensive support for ARM architectures, making it a popular choice for ARM-based embedded systems development.
- It can generate machine code for various ARM processors, from Cortex-M microcontrollers to more powerful Cortex-A application processors.
- GCC Features:
  - ✓ Cross-Compilation: GCC allows cross-compilation, enabling developers to compile code on one architecture (e.g., x86) for execution on a different architecture (e.g., ARM).
  - ✓ Optimization: GCC provides various optimization levels to improve code performance and size.
  - ✓ Debugging: It integrates with debuggers like GDB for source-level debugging.
  - ✓ Wide Language Support: GCC supports multiple programming languages, providing flexibility in language choices for embedded development.

COMPILER TOOLCHAIN

ARM TOOLCHAIN

- ARM provides a comprehensive toolchain for developing software for ARM-based processors.
- This toolchain includes not only a compiler but also other essential tools for linking, assembling, and debugging.
- ARM's toolchain is often integrated into popular Integrated Development Environments (IDEs) like Keil, Eclipse, or others, providing a user-friendly development environment.
- ARM also provides the CMSIS (Cortex Microcontroller Software Interface Standard), a standardized hardware abstraction layer for Cortex-M processors
- This includes headers, libraries, and utilities to simplify development across different Cortex-M microcontrollers.

COMPILER TOOLCHAIN

ARM TOOLCHAIN

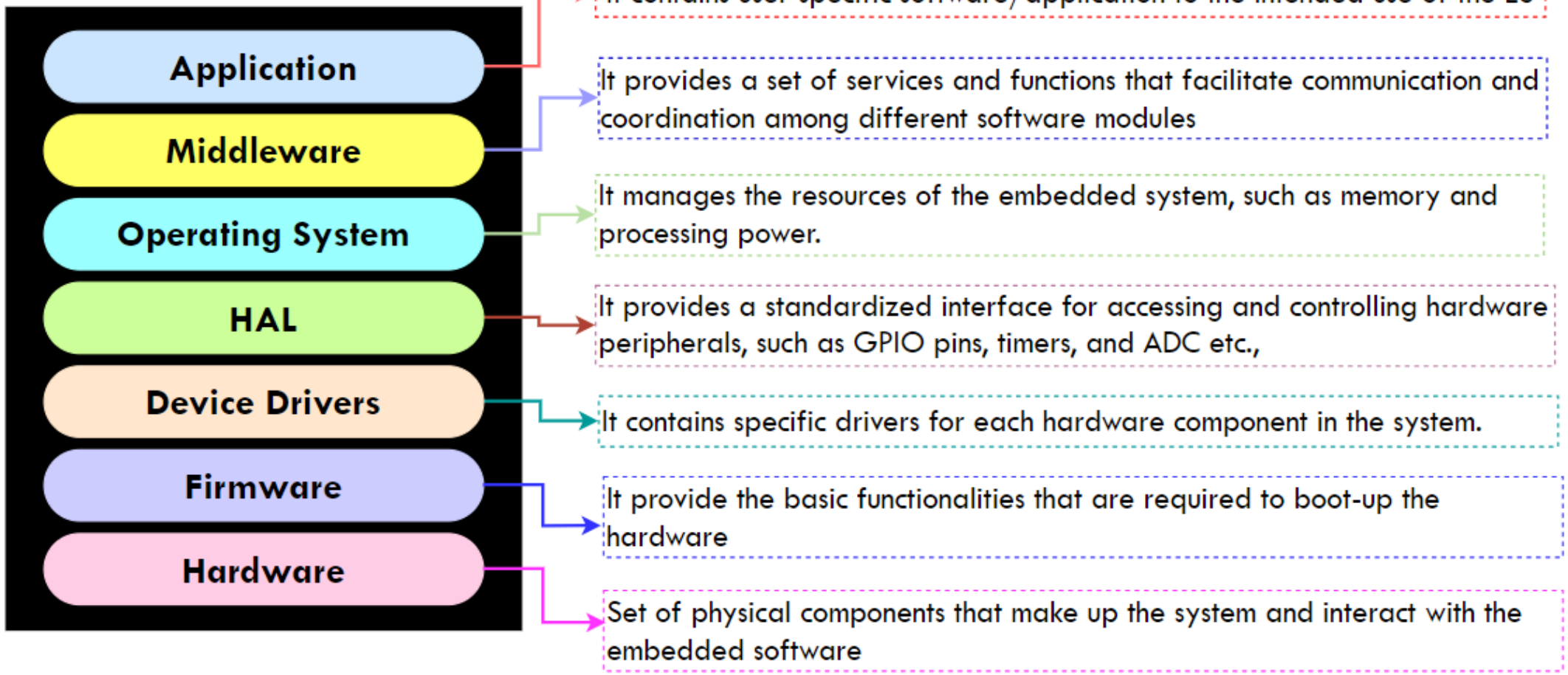
- ARM toolchain naming convention: arch [-vendor] [-os] - abi
  - arch: "arch" refers to target architecture e.g. ARM, MIPS etc.
  - vendor: "vendor" refers to toolchain supplier name e.g. apple
  - os: "os" refers to the target operating system.
  - abi: The "abi" specify which Application Binary Interface (ABI) convention is used by toolchain.

- Example: arm-none-linux-gnueabi - This toolchain targets the ARM architecture, has no vendor, creates binaries that run on the Linux OS and uses the GNU EABI.

- Components:
  - arm-none-eabi-gcc: The GCC compiler specifically configured for ARM architecture.
  - Assembler (arm-none-eabi-as): Converts assembly code into machine code.
  - Linker (arm-none-eabi-ld): Links compiled code and libraries to generate the final executable.
  - Debugger (arm-none-eabi-gdb): Allows source-level debugging of ARM code.
  - Libraries and Headers: ARM provides libraries and headers tailored for ARM architectures.

EMBEDDED SOFTWARE ARCHITECTURE

EMBEDDED SOFTWARE ARCHITECTURE



Embedded Software Architecture

EMBEDDED SOFTWARE ARCHITECTURE

- How these embedded software architecture layers interact
  - Initialization: When the embedded system is powered on, the firmware starts up first to initialize the hardware.
  - OS Booting: After hardware initialization, the firmware starts the operating system.
  - Middleware and Application Launch: Once the OS is up and running, middleware services are initialized, and finally, the application software is launched.
  - Run-Time: During operation, the application software will often make use of middleware services to accomplish its tasks. These, in turn, interact with the operating system, which may then interact with the HAL or device driver to control hardware components.
  - Shut Down: When the system is turned off, the application software is terminated first, followed by middleware services and then the OS, which may use firmware routines to safely power down the hardware.

EMBEDDED SOFTWARE ARCHITECTURE

MIDDLEWARE

- Middleware in embedded systems is a layer of software that provides additional services needed by application software, beyond what the operating system offers.
- Middleware plays a crucial role in facilitating communication, coordination, and integration among different software components and between software and hardware.
- Middleware simplifies complex actions and makes it easier for application software to perform functions. It acts like a library of functions and services that applications can call.
- Middleware acts as a bridge between application software and the OS, providing a higher-level programming interface to the developers, abstracting away the complexities related to OS and hardware interactions.
- This feature makes system integration seamless and reduces the burden of managing intricate hardware details.

EMBEDDED SOFTWARE ARCHITECTURE

MIDDLEWARE

- Middleware can be categorized into different types based on its functionality.
- Communication middleware enables communication between different components of the embedded system, while real-time middleware provides scheduling and synchronization services to ensure that tasks are executed within their deadlines.
- Middleware can also provide services for data storage, security, and fault tolerance.
- Another significant benefit of middleware is its ability to provide scalability by alllowing the addition of new components and easy scaling of the system as required, making it an indispensable component of modern embedded systems.
- Without middleware, developing and maintaining embedded systems would be significantly more challenging.

EMBEDDED SOFTWARE ARCHITECTURE

FIRMWARE

- The primary role of firmware is to provide the basic functionalities that are required to boot-up the hardware and to provide the abstraction layer on which the OS can run.
- Once the boot-up sequence is complete, passes control to the operating system.
- Firmware is the low-level software that operates a specific, single-purpose device.
- It is generally stored in read-only memory or flash storage and tightly coupled with the hardware, serving as a foundation for the operating system and other software layers.
- Many components within a PC– video adapters, disk drives, and network adapters – have firmware, as do many peripherals that connect to PC (printers, external storage etc.,)
- Efficient firmware design and execution are essential for the reliable and optimal performance of the embedded device.

EMBEDDED SOFTWARE ARCHITECTURE

DEVICE DRIVER

- Device drivers are specialized software modules that facilitate OS to communicate with and control the hardware devices.
- Device drivers are often more specific than the HAL and they are designed to interface with specific hardware, such as a network adapter, graphics card, or storage device.
- The device driver translates high-level commands from the operating system into low-level commands (with the extension .dll or .sys.) that the hardware understands.
- Device drivers are typically considered either architecture-specific or generic drivers.
- A device that is architecture-specific manages the hardware that is integrated into the master processor (the architecture). Examples include on-chip memory, integrated Memory Managing Units (MMUs) and floating-point hardware.

EMBEDDED SOFTWARE ARCHITECTURE

DEVICE DRIVER

- A generic device driver manages hardware that is located on the board and not integrated into the master processor. Examples: I2C, PCI, off-chip memory, off-chip I/O's.

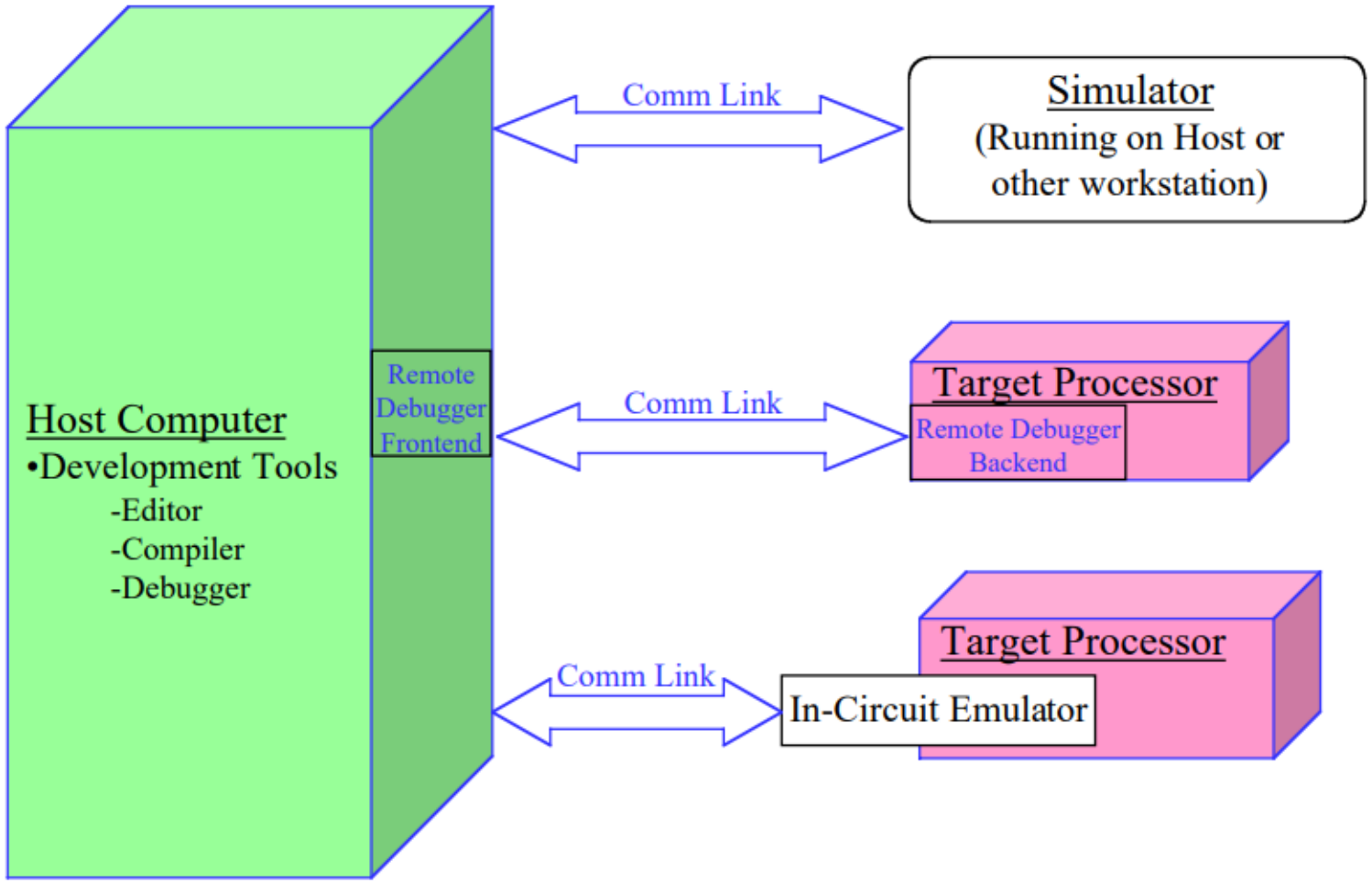
- Regardless of the type of device driver, it performs functions like:
  - ✓ Startup/ Shutdown of Hardware allows the initialization of the hardware upon power-on or reset or configuring hardware into its power-off state.
  - ✓ Disable/Enable of Hardware allows other software to disable or enable the hardware.
  - ✓ Release/Aquire of Hardware allows other software to free (unlock) the hardware or to gain singular (locking) access to hardware.
  - ✓ Read/Write of Hardware allows other software to read/write data from the hardware.
  - ✓ Install/Uninstall of Hardware allows other software to install new hardware or to remove hardware.

DEBUGGING TOOLS

DEBUGGING TOOLS

- Debugging plays a vital role in embedded systems development which helps to ensure the proper functioning of embedded systems by:
  - Locate and resolve software bugs and hardware issues.
  - Optimizing resource usage, which is crucial in systems with limited resources.
  - Improve system performance and efficiency.
  - Enhancing system stability and reliability.
  - Ensure compliance with industry standards and best practices.
- Using debugging techniques/tools, embedded systems developers can create high-quality, reliable, and efficient systems that meet the demands of various industries.
- Some of the commonly used debugging tools for embedded systems: Simulators, IDEs, Emulators, In-Circuit Debuggers, Logic Analyser.

DEBUGGING TOOLS



DEBUGGING TOOLS

SIMULATORS

- Simulator create a virtual models of embedded systems in software environments, allowing developers to test and debug their code without the need for physical hardware.
- Benefits of simulator include reduced development costs, faster debugging cycles, and the ability to test various scenarios and configurations.
- A simulator performs the following functions
  - ✓ Defines the processor family as well as its various versions for the target system.
  - ✓ Provides the detailed information of a source code, status of RAM and simulated ports of the target system for each single step execution.
  - ✓ Monitors system response and determines throughput.
  - ✓ Provides trace of the output of contents of program counter versus the processor registers.
  - ✓ Supports the conditions and unconditional breakpoints.
- Disadvantage of simulator is that, it can't simulate the function of all peripherals.

DEBUGGING TOOLS

SIMULATORS

Keil µVision IDE

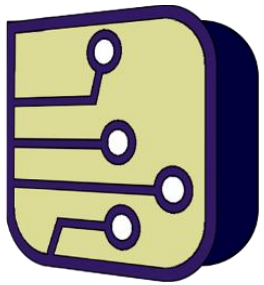
IAR Embedded Workbench

STM32 Cube IDE

MPLAB X IDE

Popular IDEs for basic MCU simulation

Proteus Design Suite



SimulIDE

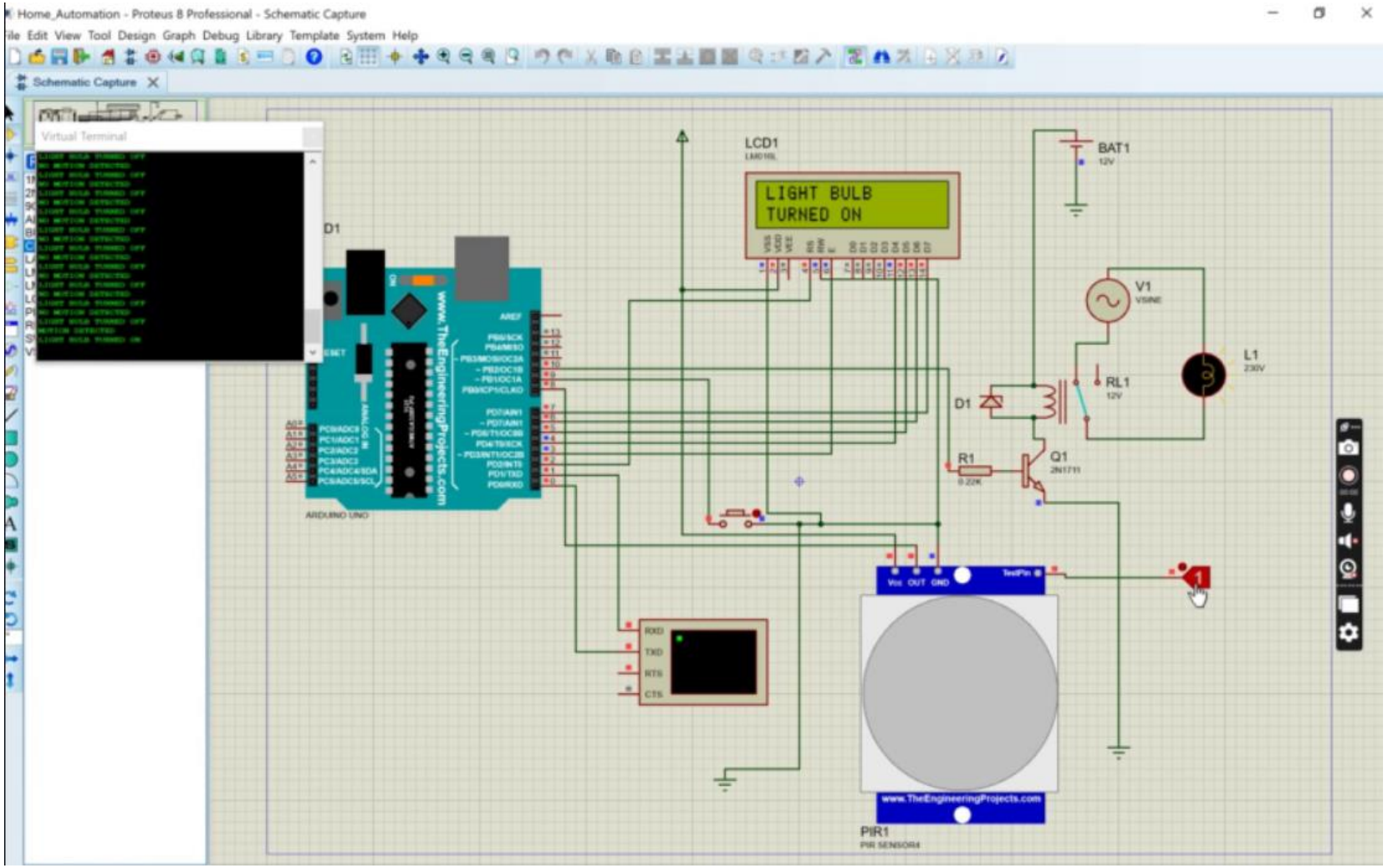
TinkerCAD circuits

Wokwi

Popular simulators for system-level simulation

DEBUGGING TOOLS

SIMULATORS



Screenshot of the Proteus Software interface for Home Automation system

DEBUGGING TOOLS

IN-CIRCUIT DEBUGGER (ICD)

- An in-circuit debugger(ICD) is a hardware device used to facilitate the testing and debugging of microcontroller-based real-time applications faster and easier.
- ICD is connected between a PC and the target microcontroller system, and allows developers to examine and modify the program running on a microcontroller via communication link while it is still connected to the target hardware.
- During debugging, the ICD device adds an additional block of code (along with the user application) to the target microcontroller's program memory, which interacts with the debugging application in PC through the same programming cable.
- The program running on the host of a ICD has a user interface (GUI/Command-line) to show the active part of the source code, current register contents, and other relevant information about the executing program.

DEBUGGING TOOLS

IN-CIRCUIT DEBUGGER (ICD)

- Using ICD, the programmer can perform the following,
  - Start/restart/kill, and stepping through program.
  - Set Software breakpoints.
  - Reading/writing registers or data at specified address.
- ICD also include programming functions that enable the target microcontroller to be programmed through the same pins that are used for In-Circuit Serial Programming (ICSP).
- Using an ICD significantly speeds up the development and debugging process by allowing developers to identify and fix issues directly on the target hardware.
- Disadvantage: Inability to debug startup Code, code must execute from RAM, requires a target processor to run the final software package

DEBUGGING TOOLS

IN-CIRCUIT DEBUGGER (ICD)

ST LINK V2 in-circuit Debugger / Programmer for STM8 and STM32

MPLAB PICkit 5 In-Circuit Debugger/Programmer for All Microchip Devices

IAR Systems I-Jet Power Debugger

Popular In-Circuit Debuggers

DEBUGGING TOOLS

IN-CIRCUIT EMULATORS (ICE)

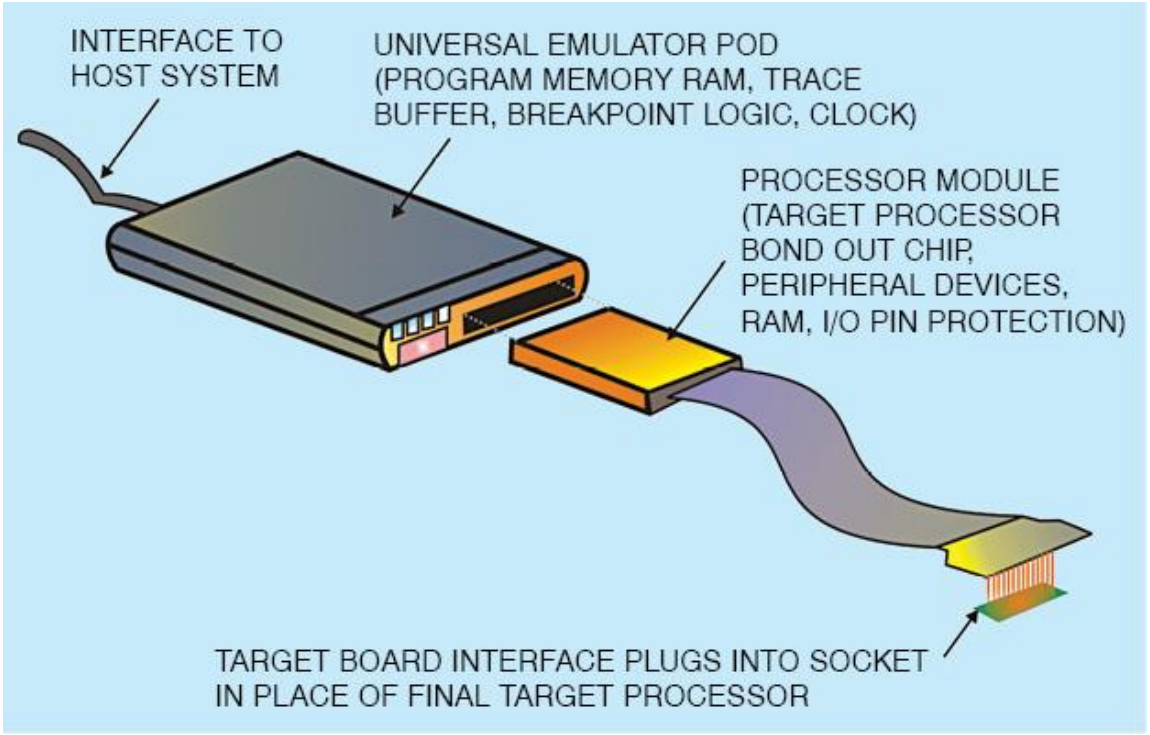
- In embedded systems, an emulator is a replication of the target system (Microcontroller) with identical functionality and components.
- It contains a copy of target processor, plus RAM, ROM, and its own embedded software.
- This tool is needed to analyse performance of the code and to see how the code will work in the real-time environment.
- This is particularly useful during the development and testing phases when the actual hardware may not be readily available or to speed up the development process.
- Using emulators, programmers can change values in order to reach the ideal performance of the code. Once the code is fully checked, it can be uploaded into the device.

DEBUGGING TOOLS

IN-CIRCUIT EMULATORS (ICE)

- The microcontroller program is downloaded into the emulation RAM and the emulation processor executes instructions using the same data registers and peripherals as the target processor.

- The I/Os of the emulation processor are made available on a socket that is plugged into the system under development instead of the target processor being emulated.



In-circuit emulator

DEBUGGING TOOLS

IN-CIRCUIT EMULATORS (ICE)

- Few ICE have special ASICs or FPGAs that imitate core processor code execution and peripherals, but there may be behavioral differences between the actual device and the emulator.
- State-of-the-art emulators offer real-time code execution, full peripheral implementation, multilevel conditional breakpoints capability, real-time trace buffers, and some will time-stamp instruction execution for code profiling.
- In addition to providing the features available with a debugger, an ICE allows programmer to debug startup code and programs running from ROM, etc.,
- ICE provides greater flexibility, ease for developing various applications on a single system in place of testing that multiple targeted systems but the disadvantage is that, it is expensive.

DEBUGGING TOOLS

IN-CIRCUIT EMULATORS (ICE)

SEGGER J-LINK-ARM V8 USB-JTAG Adapter Emulator

Analog Devices ADZS-ICE-2000 EMULATOR BLACKFIN

Silicom labs C8051F MCU Emulator U-EC6 USB

Popular In-Circuit Emulators

DEBUGGING TOOLS

LOGIC ANALYSER

- Hardware debugging involves diagnosing and fixing issues related to the physical components of an embedded system, such as circuitry, sensors, and actuators.
- Benefits of hardware debugging include improved system reliability, reduced development time, and the ability to identify and resolve hardware-specific issues.
- Some popular hardware debugging tools for embedded systems include:
  - Oscilloscopes: Essential tools for analysing and troubleshooting electrical signals
  - Logic Analysers: Devices used for monitoring and analysing digital signals
  - Protocol Analysers: Tools for capturing and analysing communication data
  - Power Analysers: Instruments for measuring and analysing power consumption
- Among above listed hardware debugging tools, logic analyser is most commonly used hardware debugging tool in many embedded system applications.

DEBUGGING TOOLS

LOGIC ANALYSER

- Logic analyser is a powerful tool which physically interfaces to the hardware signal lines to capture and analyse the digital signals and data exchanged between your microcontroller and other devices, such as sensors, memory etc.,

- Some factors to consider best logic analyser are the number of channels, the sampling rate, the memory depth, the trigger options, and the software compatibility.

- Logic analysers are mainly used to
  - ✓ Debug and verify digital system operation
  - ✓ Trace and correlate many digital signals simultaneously
  - ✓ Detect and analyse timing violations and transients on buses
  - ✓ Trace embedded software execution
  - ✓ Measure the timing of the power up and system initialize sequence

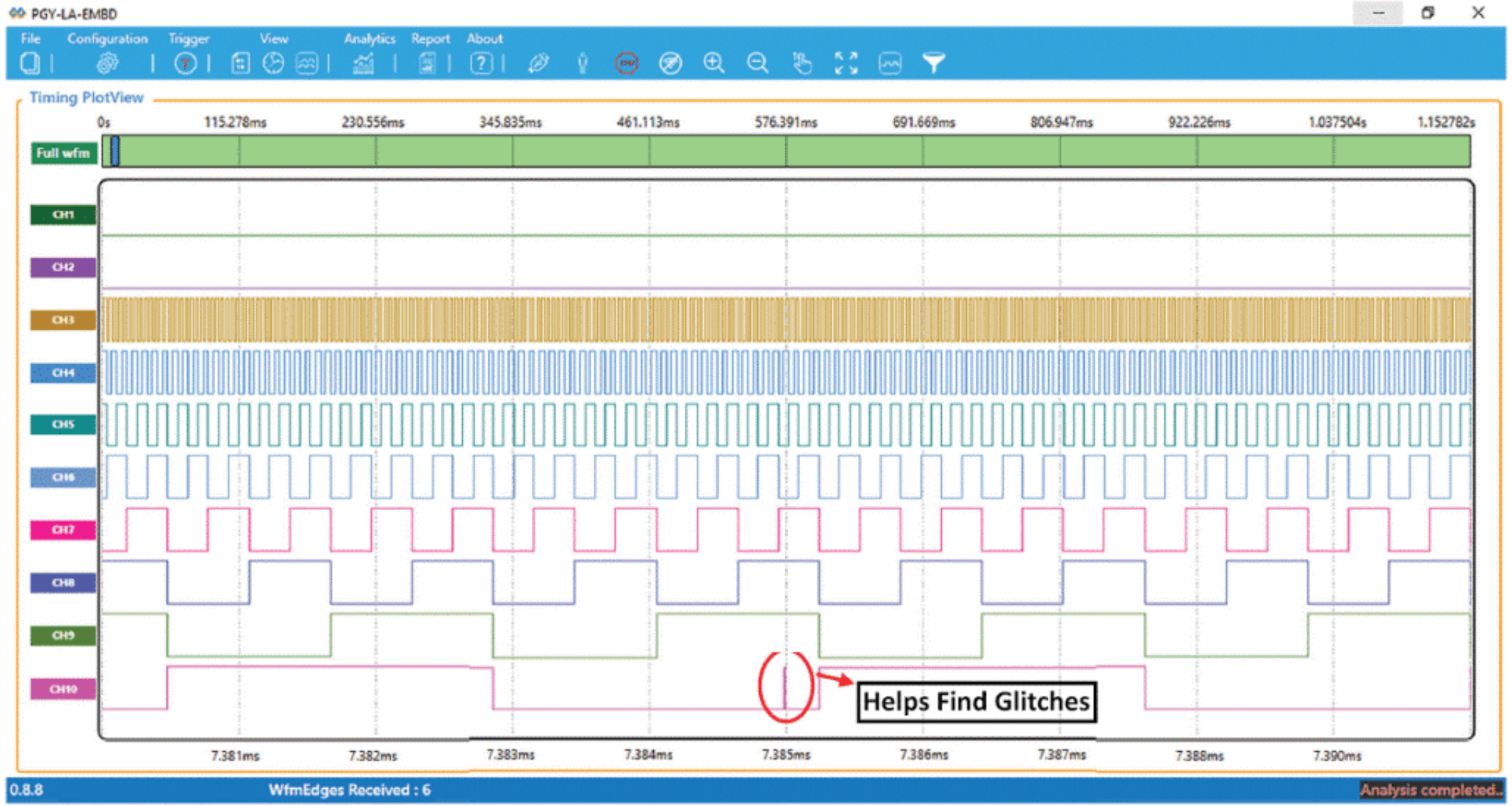
DEBUGGING TOOLS

LOGIC ANALYSER

- Steps to use a logic analyser to debug embedded software:
  - Step 1: Choose a logic analyser
  - Step 2: Connect the logic analyser to your circuit
  - Step 3: Set up the logic analyser software
  - Step 4: Run your embedded software and capture the data
  - Step 5: Analyse the captured data
  - Step 6: Debug and fix your embedded software

DEBUGGING TOOLS

LOGIC ANALYZER



Software interface for the Discovery series logic analyser from Prodigy Technovations

DEBUGGING TOOLS

LOGIC ANALYZER

Prodigy Technovations
Discovery series logic analyser

Logic Pro Saleae 8-
Channel Logic Analyser

Nordic Semiconductor Power
Profiler Kit II (PPK2)

InnoMaker LA1010
Kingst Logic Analyser

Digilent Digital Discovery
USB Logic Analyzer

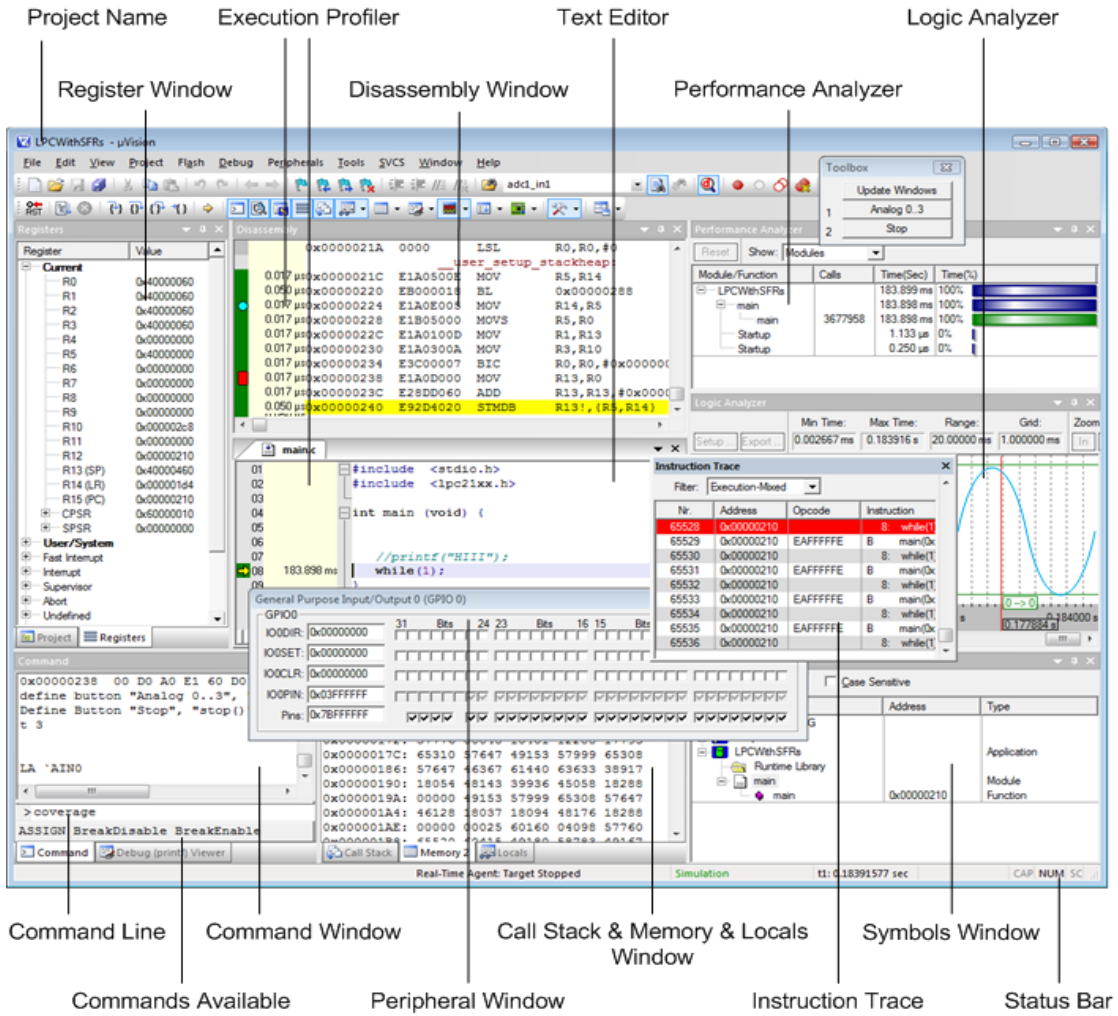
DEBUGGING TOOLS

INTEGRATED DEVELOPMENT ENVIRONMENT (IDE)

- An Integrated Development Environment (IDE) for embedded systems is a comprehensive software tool that provides a unified environment for embedded software development.
- It brings together various features and tools needed by developers to write, compile, debug, and deploy software for embedded platforms.
- Key features and components often found in IDEs for embedded systems: Code Editor, Compiler and Toolchain Integration, Debugger, Project Management etc.,
- Popular IDEs for embedded systems include Keil µVision, IAR Embedded Workbench, Eclipse with various plugins, and STM32Cube IDE.
- The choice of IDE often depends on the target hardware, project requirements, and developer preferences.

DEBUGGING TOOLS

INTEGRATED DEVELOPMENT ENVIRONMENT (IDE)



Screenshot of Keil IDE

THANK YOU