Fundamentals of Embedded and Real Time Systems

MODULE 02

TAMER AWAD

Review Module 01

Module 02

Embedded development environment

- IDE
- IAR
- Common views
- Demo 01 – IDE usage
- Compilers and Tool Chains
- Debugging and Testing

Evaluation Board

- Evaluation board overview
- Running a program on embedded board

C Files

- Preprocessor directives
- Conditional Compilation

C Environnent

- File Structure
- Preprocessor
- Conditional Compilation

C Programming

- Variables
- Keywords
- Operators
- Flow control
- Demo 02 – Flow control
- Pointers
- Data Types
- Demo 03 – Variables & Pointers

Embedded Development Environment overview

- IDE
- IAR
- Common views
- Demo 01
- Compilers and Tool Chains
- Debugging and Testing

Integrated Development Environment (IDE)

1

Editor you use to write program

2

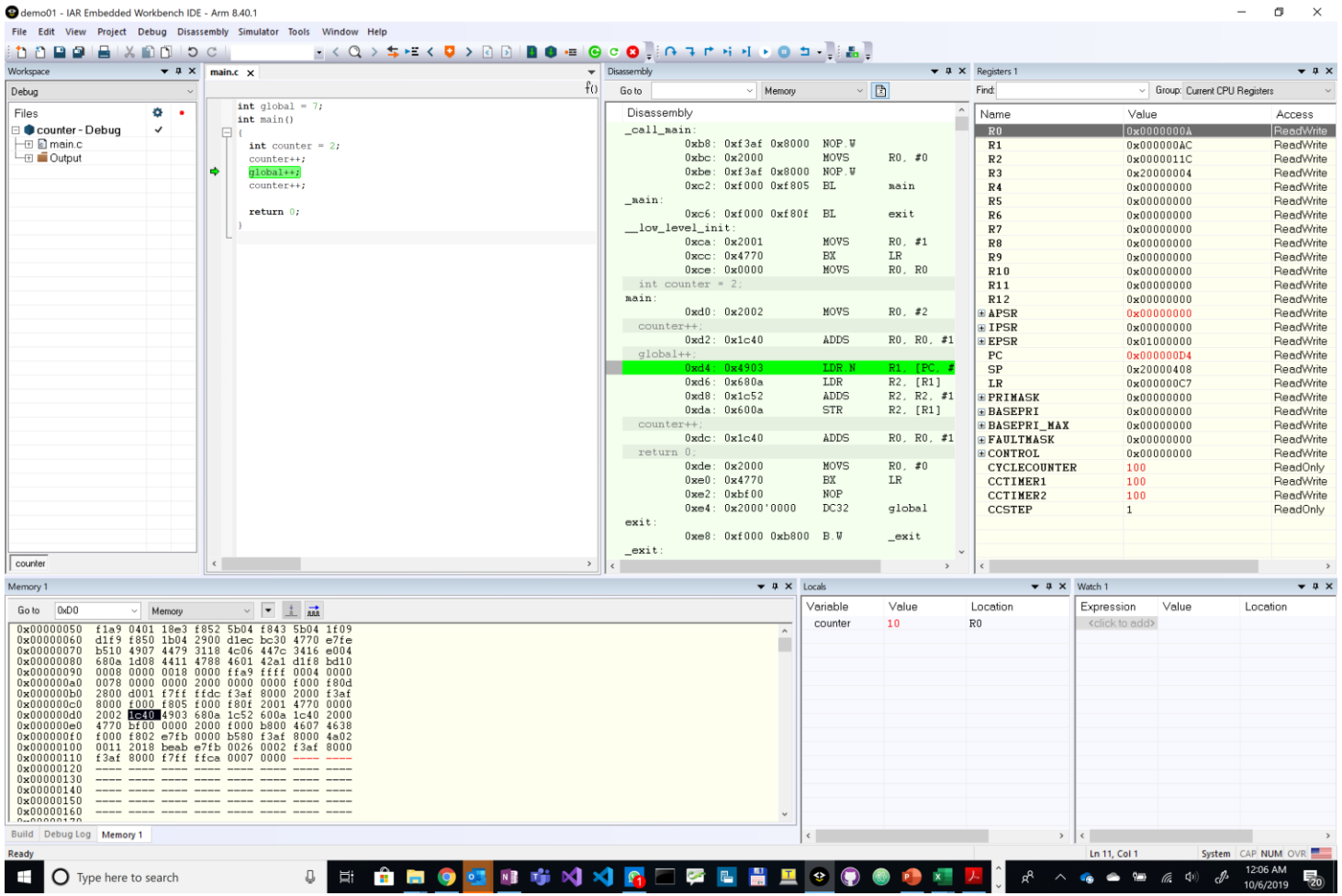
Compiler convert code to binary file that is readable by the processor

3

Load compiled code to flash

4

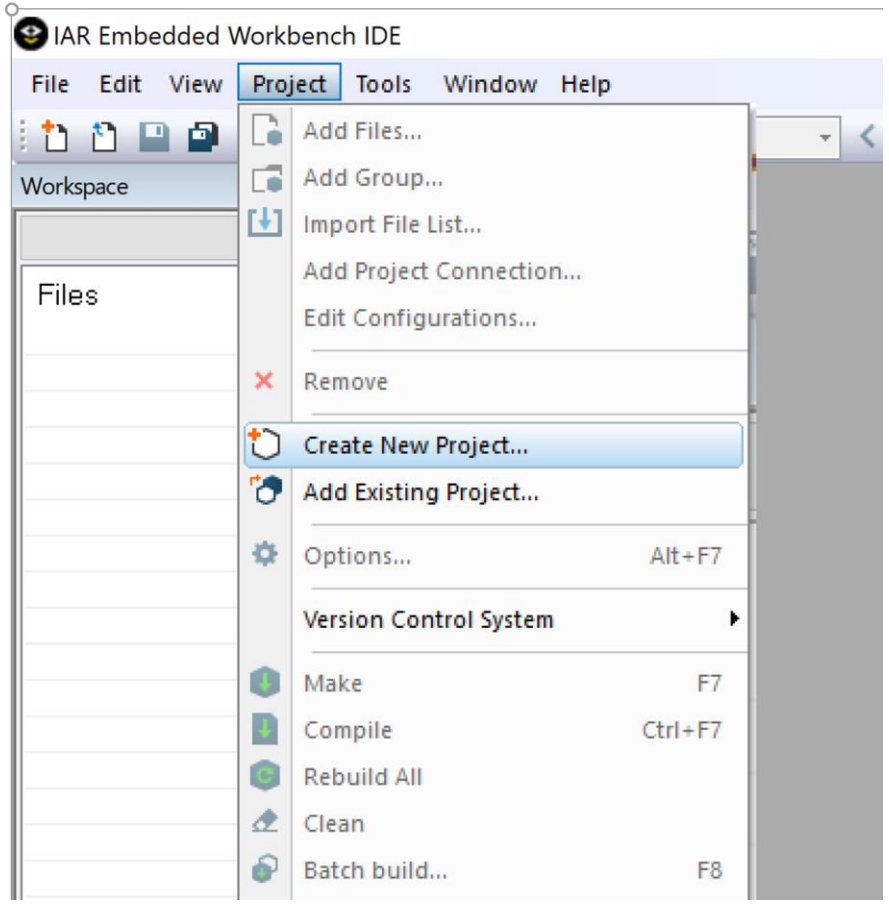
Instruction set simulator to run the program on the simulated environment as if running on the real processor

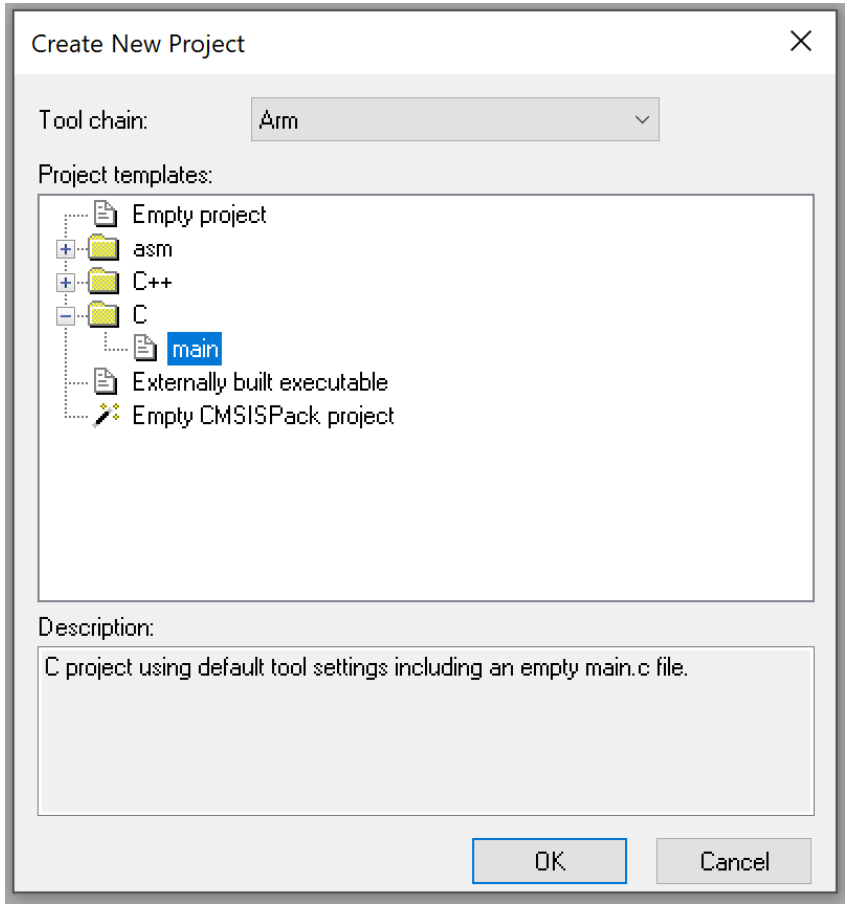


IAR Embedded Work Bench

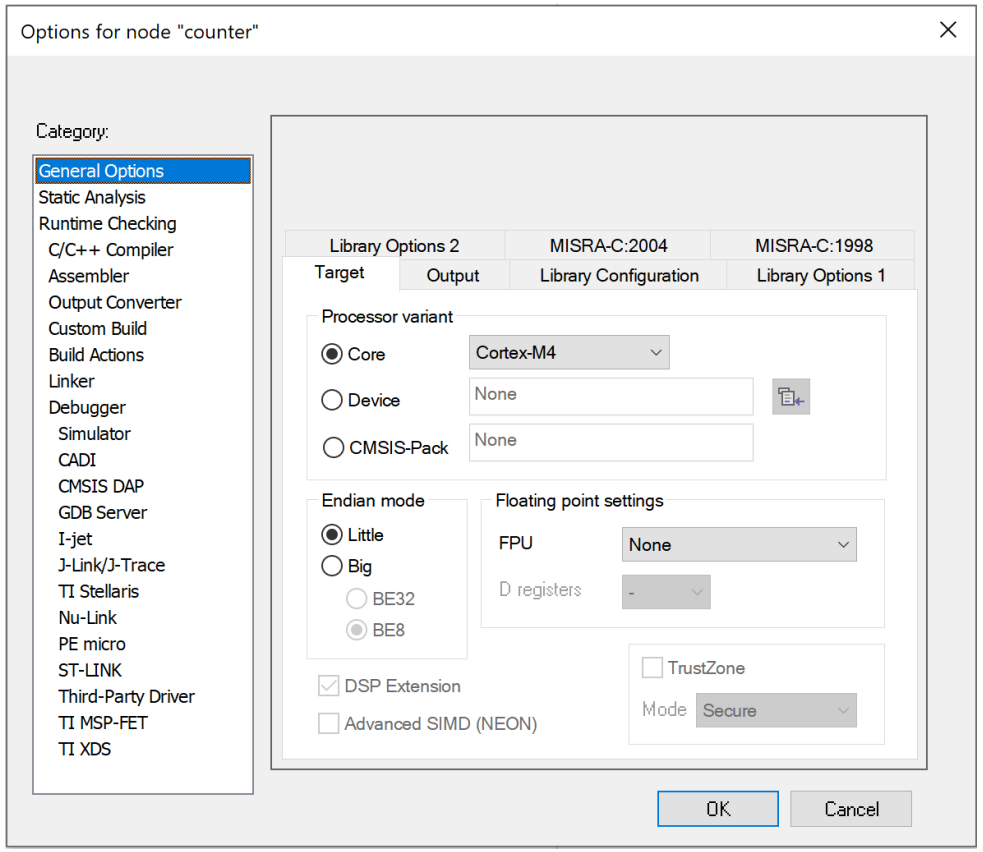
IDE

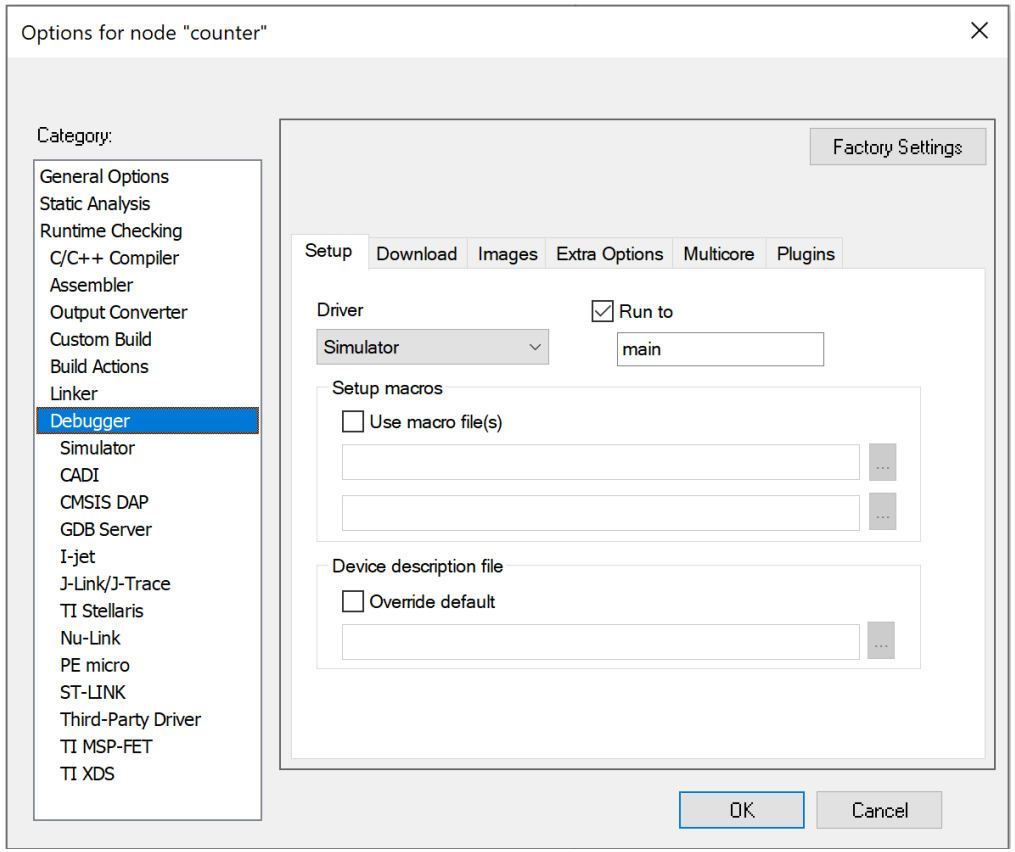
Create New Project



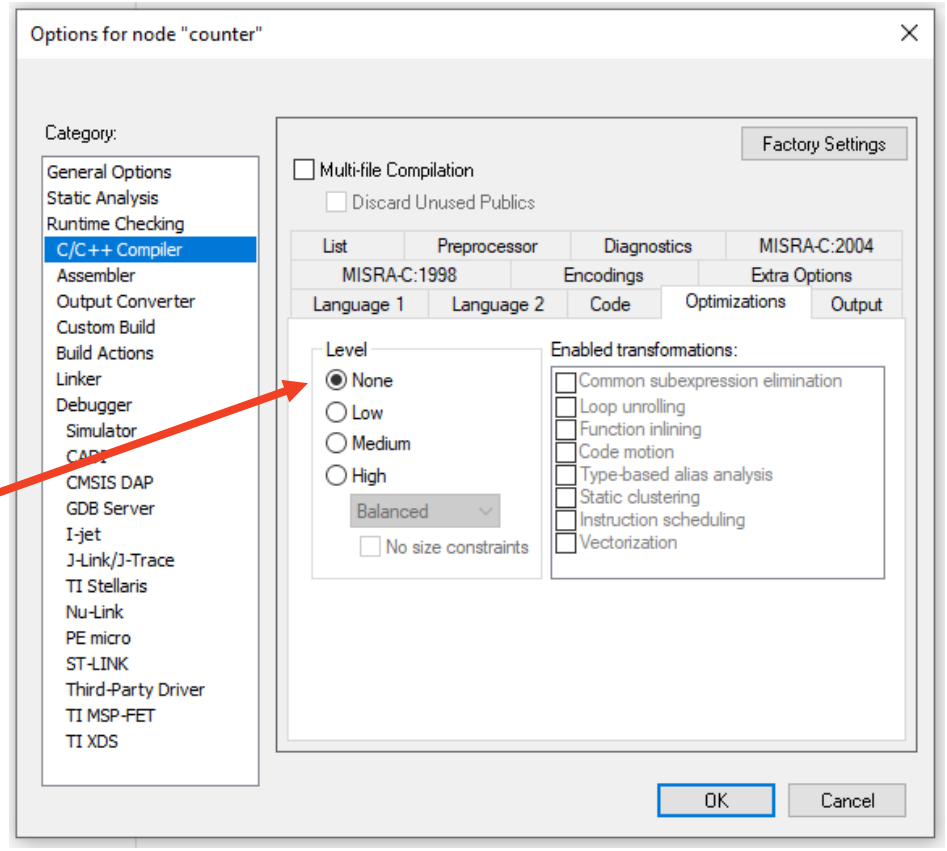


Project --> Options

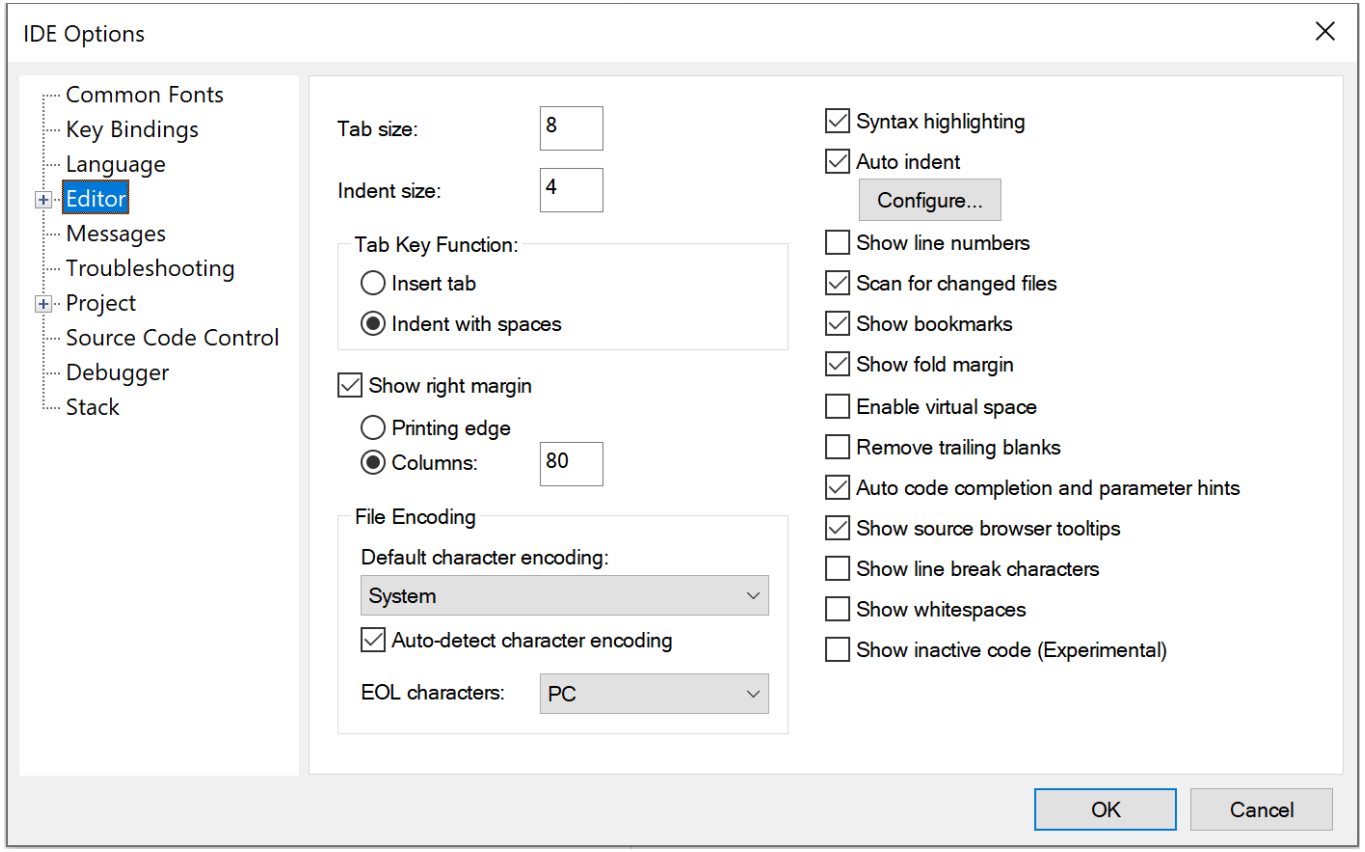




Project --> Options



IDE Options


IDE Options
Common Fonts
Key Bindings
Language
Editor
Messages
Troubleshooting
Project
Source Code Control
Debugger
Stack
Tab size: 8
Indent size: 4
Tab Key Function:
Insert tab
Indent with spaces
Show right margin
Printing edge
Columns: 80
File Encoding
Default character encoding:
System
Auto-detect character encoding
EOL characters: PC
Syntax highlighting
Auto indent
Configure...
Show line numbers
Scan for changed files
Show bookmarks
Show fold margin
Enable virtual space
Remove trailing blanks
Auto code completion and parameter hints
Show source browser tooltips
Show line break characters
Show whitespaces
Show inactive code (Experimental)
OK
Cancel

Demo - 01

- Create project from scratch
- Explore IDE editor options
- Compiler is your friend
- Use counter example
- Explain variables
- Setup simulator
- Open debug views (disassembly, register, local, memory)
- Explore disassembly view
- Explore stepping thru program
- Explore CPU registers (Generic registers+ PC register)
- MOV & ADD instructions

Hello “counter” example

```
int main()
{
  int counter = 2;
  counter++;
  counter++;
  counter++;
  counter++;
  counter++;
    return 0;
}
```

Running IAR – Debug with simulator

- Allows execution of one instruction at a time, so you can observe registers and memory that are changing
- We'll be using the simulator for ARM Cortex M4 processor

IAR - Views

- Disassembly
- Register
- Memory
- Locals
- Watch

IAR – Disassembly View

- Each instruction has an address
- Machine instructions are just binary numbers
- Mnemonics added by IAR for readability
  - "A **mnemonic** is a symbolic name for a single executable machine **language** instruction (an opcode)",
    - Source: https://en.wikipedia.org/wiki/Assembly_language

IAR - Registers

- What is a register?
  - Very fast memory
  - Associated with the processor
- ARM has 16x **32-bit** registers
- Machine instructions manipulate registers directly (typically in one clock cycle)
- What are their purpose?
  - R0 – R12: general purpose
  - R13 – Stack pointer: used when calling a function
  - R14 – Link register: used when returning from a function
  - R15 – Program counter: points to the current instruction

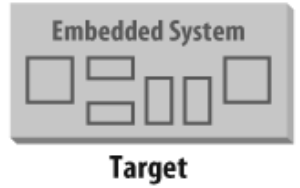
IAR – Memory View

- Big table of bytes
- Address – numbered sequentially
- 0x00000000 → 0xFFFFFFFF
- Memory regions – appear in the same space, in different area access with different buses
  - Executable space
  - Global data space
  - Shared memory space
  - Peripherals

Host and Target Development

- Embedded software development is typically done on a host machine.
  - **Host** - a computer on which you are compiling
- This is different from the target machine on which the software will eventually be run.
  - **Target** - the computer on which you run the code

The development tools that build the embedded software run on a general-purpose computer



The embedded software that is built by those tools runs on the embedded system

Compilers, assemblers, and linkers

- **A compiler:**
  - Converts high level language into machine code
- **Assembler**
  - Converts assembly language into machine code
- A **native-compiler**
  - A compiler where the target and the host are the same
- A **cross-compiler**
  - A compiler where the target is different from the host.
  - Understands the same C language as a native compiler (with a few exceptions), but its output uses the instruction set of the target microprocessor.
- A **cross-assembler**
  - Understands an assembly language that is specific to your target microprocessor and outputs instructions for that microprocessor.
- A **linker/locator**
  - Combines separately compiled and assembled modules into an executable image.
  - In addition, it places code, data, startup code, constant strings, at suitable addresses in ROM and RAM.

What is a tool chain?

- A toolchain is a set of tools containing:
  - Compiler
  - Assembler
  - Linker
  - Shared libraries
  - A method for loading the software onto the target machine
  - Any other tools you need to produce the executable for the target
  - A debugger and/or IDE may also count as part of a toolchain

Native-Compilers

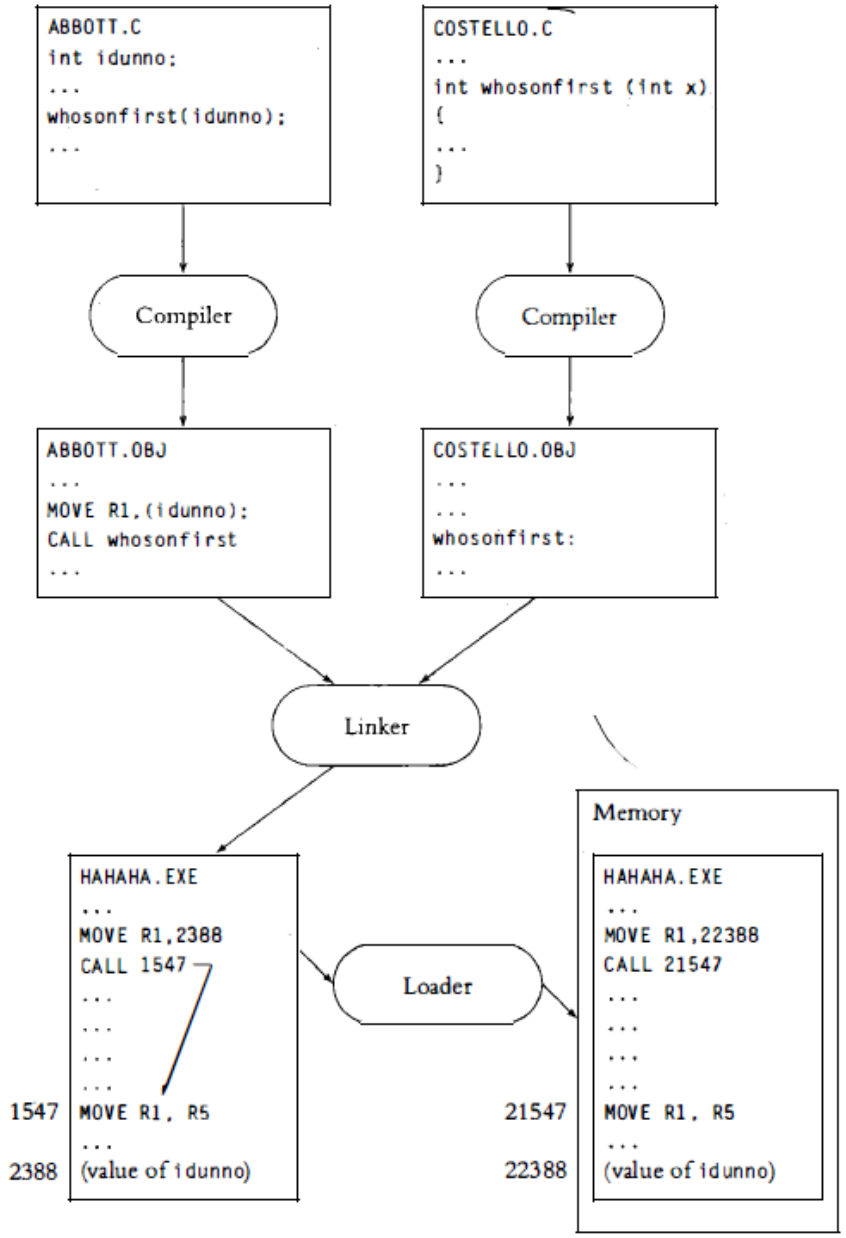


Diagram credit: David Simon

Cross-Compilers

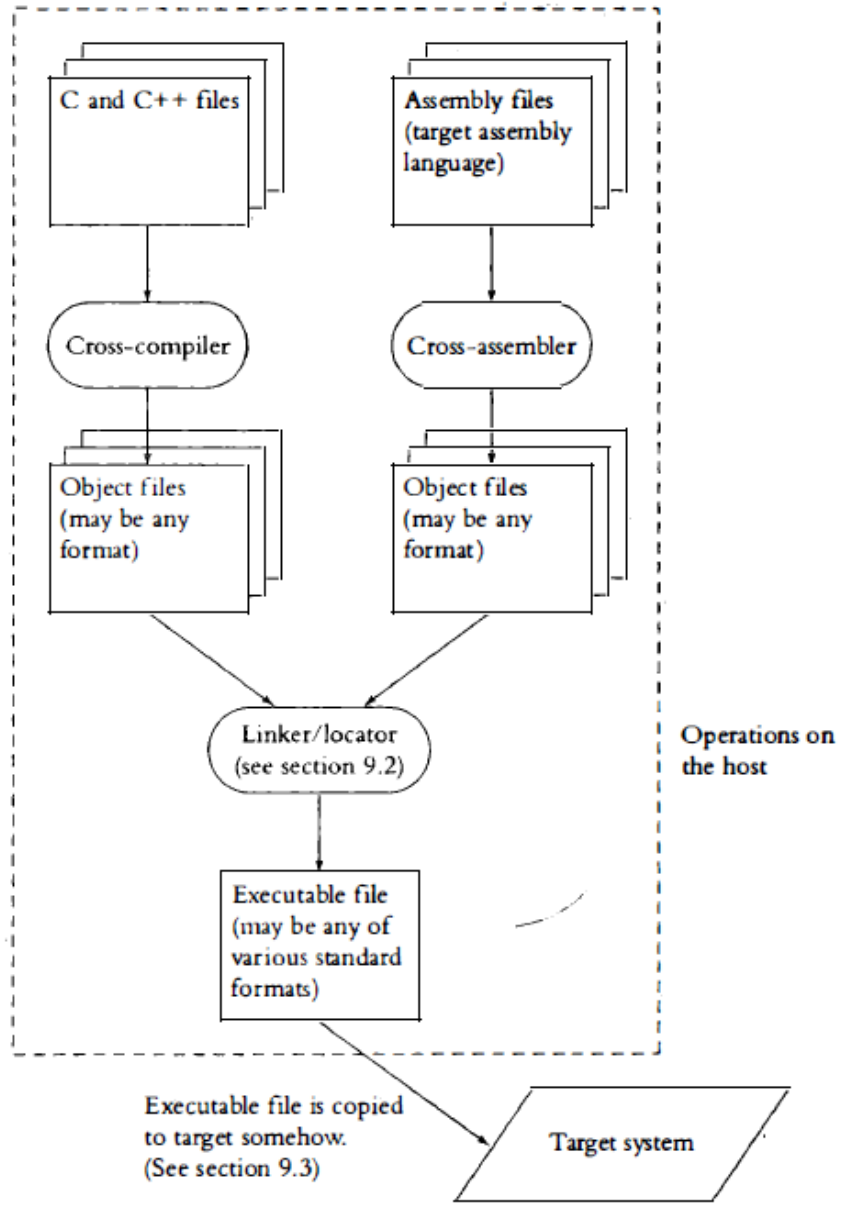
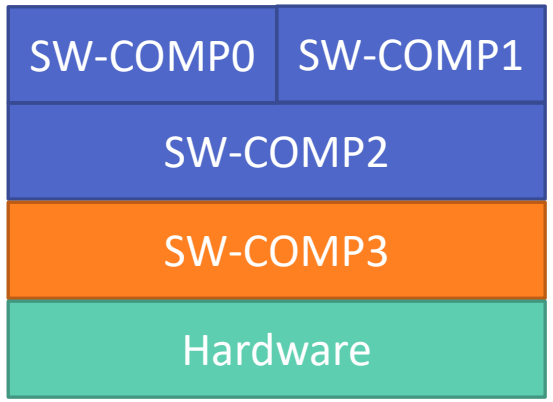


Diagram credit: David Simon

Debugging and testing

- Remember that the host system is a much friendlier environment for testing and debugging.
- Test as many layers of your software on your host machine.
- “Mock” hardware dependent layers in your testing
- A layered design makes for easy testing, debugging and “porting”.

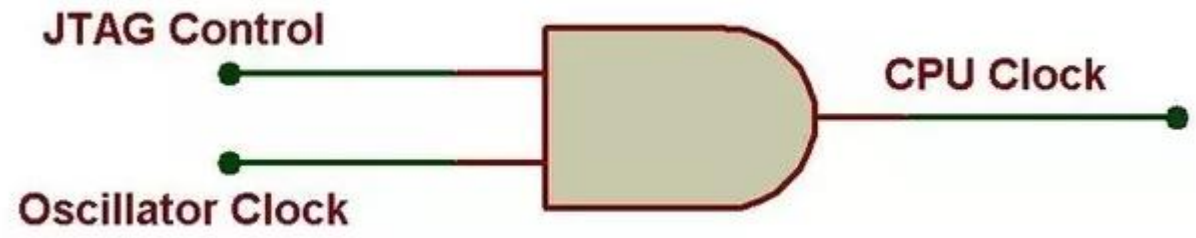


Hardware agnostic
Hardware dependent

Debugging tools

- **JTAG** (Joint Test Action Group)
- 5 Wire protocol (TRST, TCLK, TDI, TDO, TMS)
- An interface used for debugging and programming devices like microcontrollers, ASICs, CPLDs, FPGAs etc.
- Enables debugging the hardware in real time.
- Controls directly the clock cycles provided to the micro controller through software

- Serial Wire Debug (**SWD**) is a 2-pin (SWDIO/SWCLK) electrical alternative **JTAG** interface that has the same **JTAG** protocol on top.
- The ST-LINK/V2 is an in-circuit debugger/programmer for the STM8 and STM32 microcontrollers.



Credit to https://www.quora.com/What-is-JTAG

BREAK 1

Evaluation board

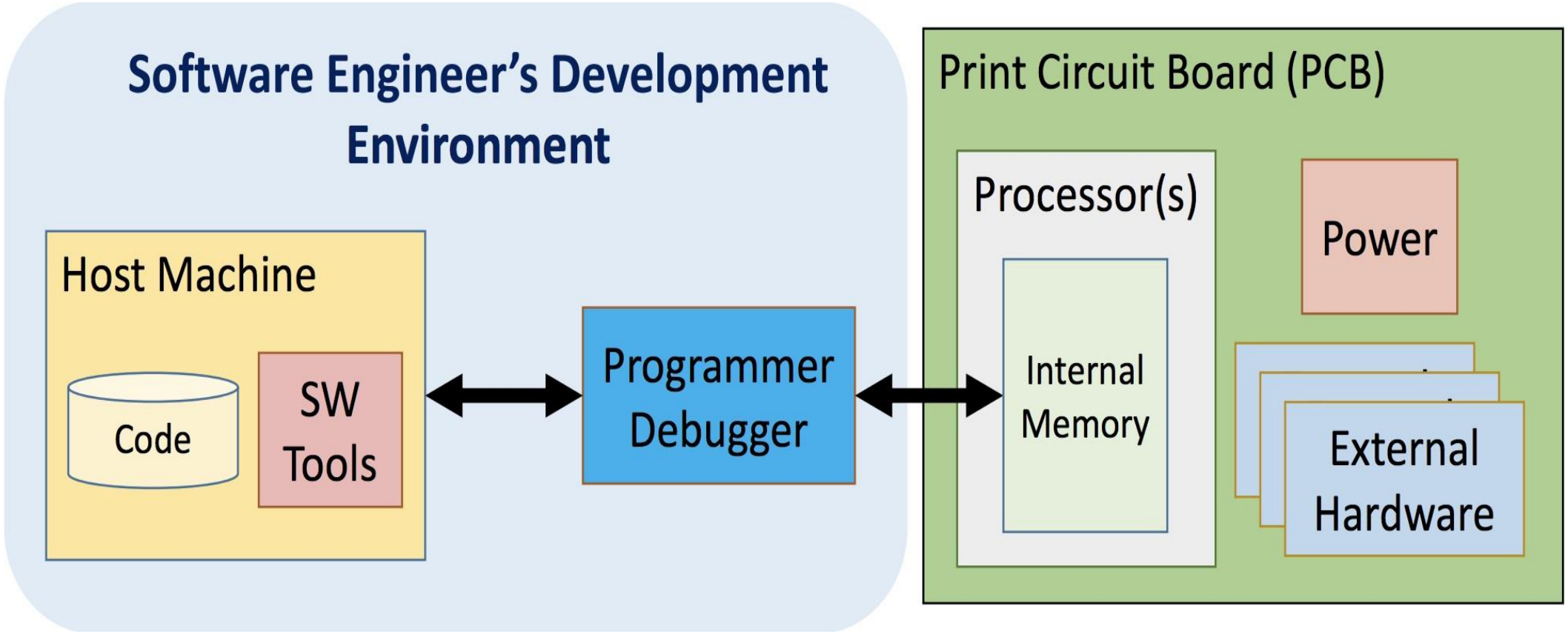
- Evaluation board overview
- Running a program on embedded board

Evaluation Board

- A printed circuit board containing a **microprocessor/microcontroller** and the **minimal support logic** needed for an engineer to become acquainted with the chip on the board and to learn to how to program it.

- Also serves as a method to prototype applications in products.

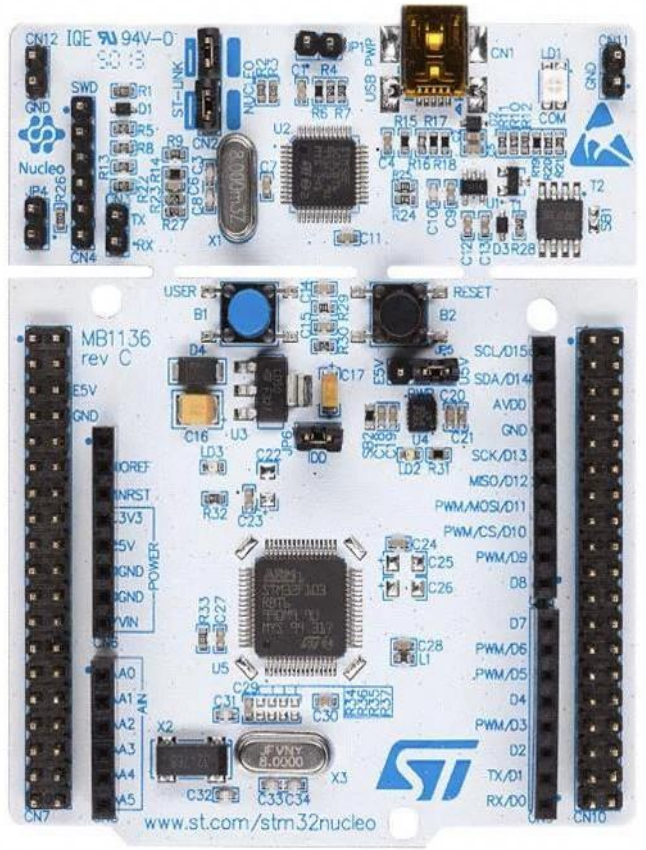
Evaluation board



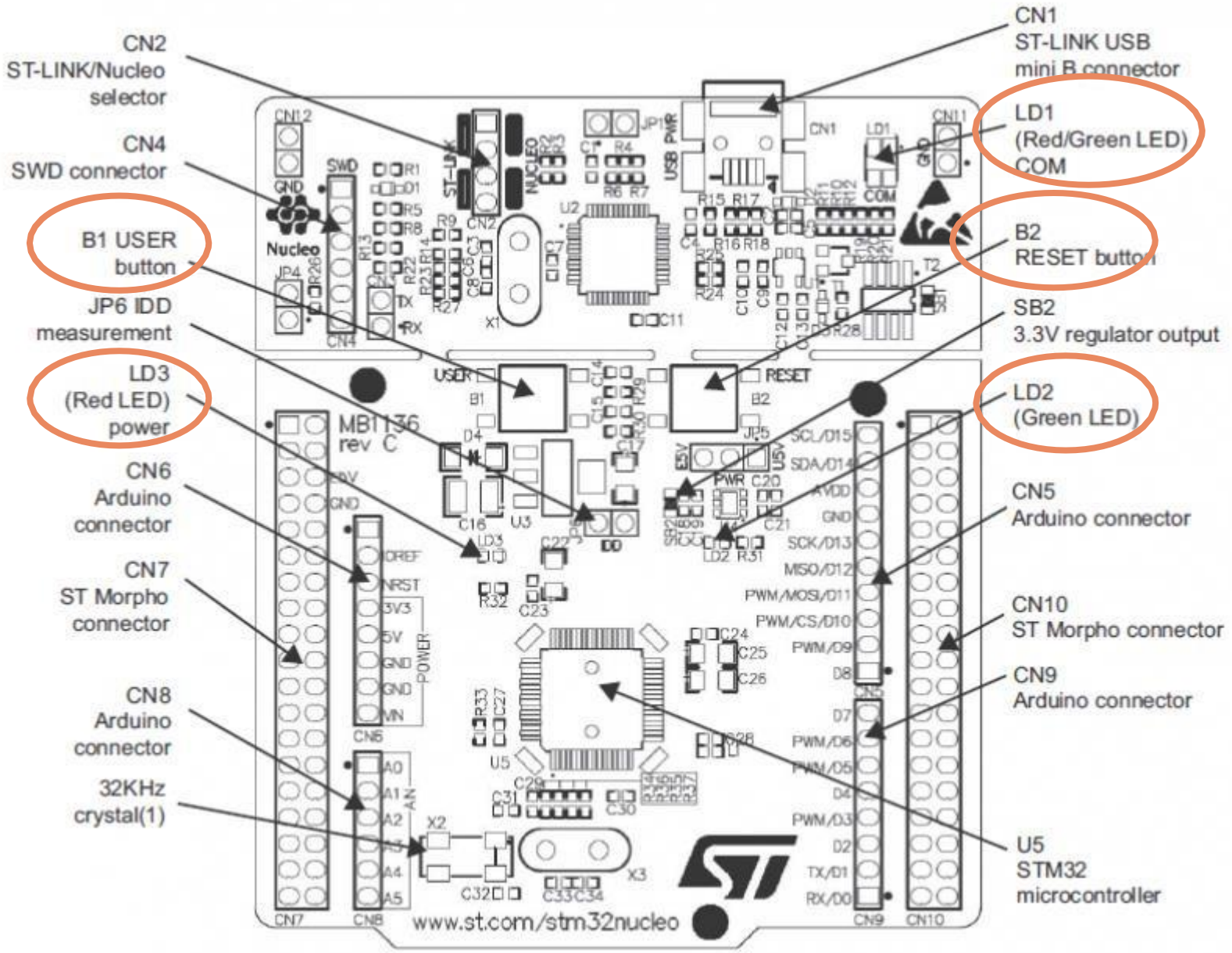
Evaluation Board - STM32F401xE Nucleo

Key Features:

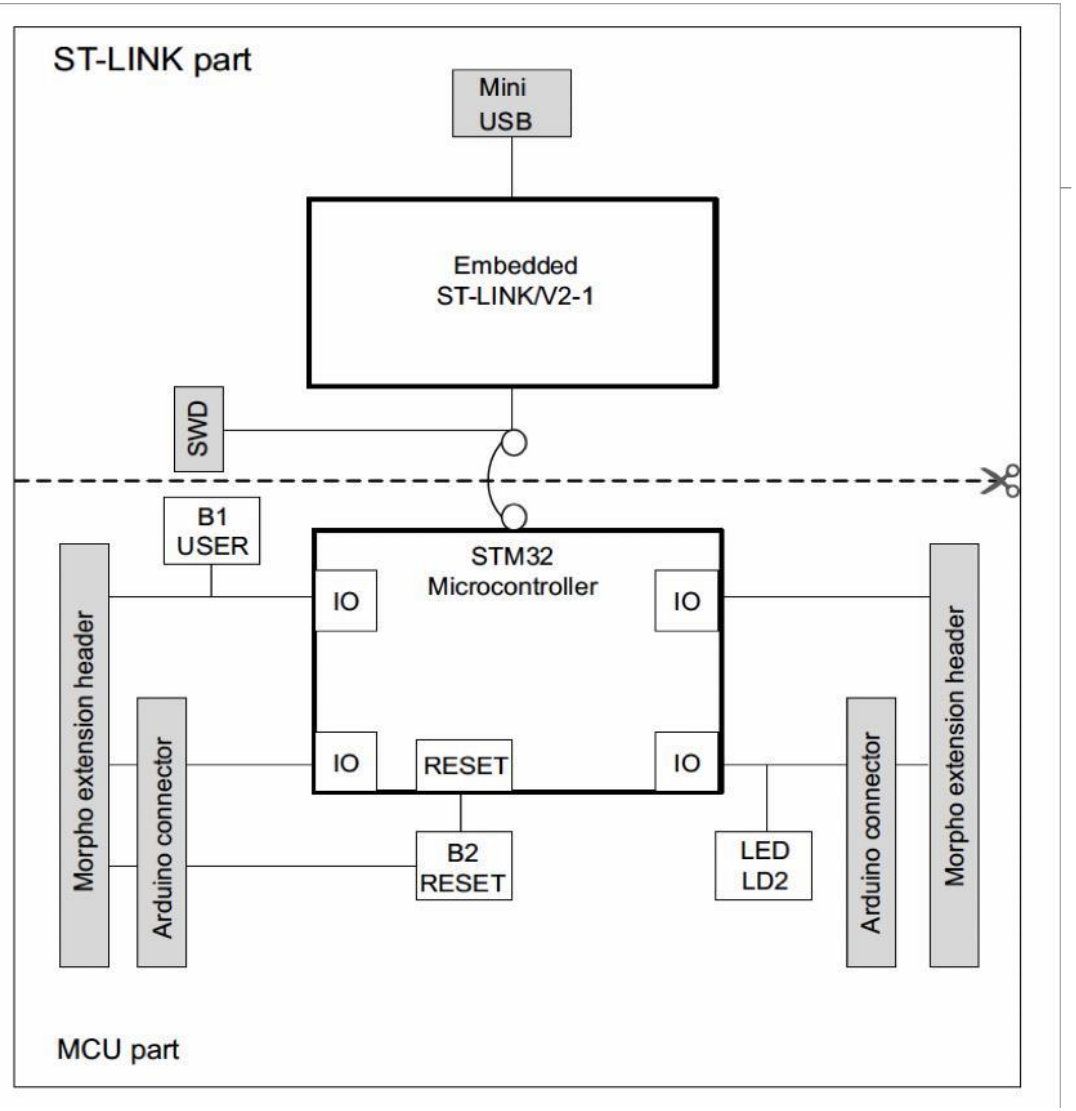
- STM32 microcontroller in LQFP64 package
- 1 user LED
- 1 user and 1 reset push-buttons
- 32.768 kHz crystal oscillator
- Board connectors:
  - Arduino™ Uno V3 expansion connector
  - ST morpho extension pin headers for full access to all STM32 I/Os
- On-board ST-LINK debugger/programmer
- Comprehensive free software libraries and examples available with the STM32Cube MCU Package
- Support of a wide choice of Integrated Development Environments (IDEs) including IAR™, Keil® and GCC-based IDEs



Evaluation Board - STM32F401xE Nucleo



Evaluation Board - STM32F401xE Nucleo

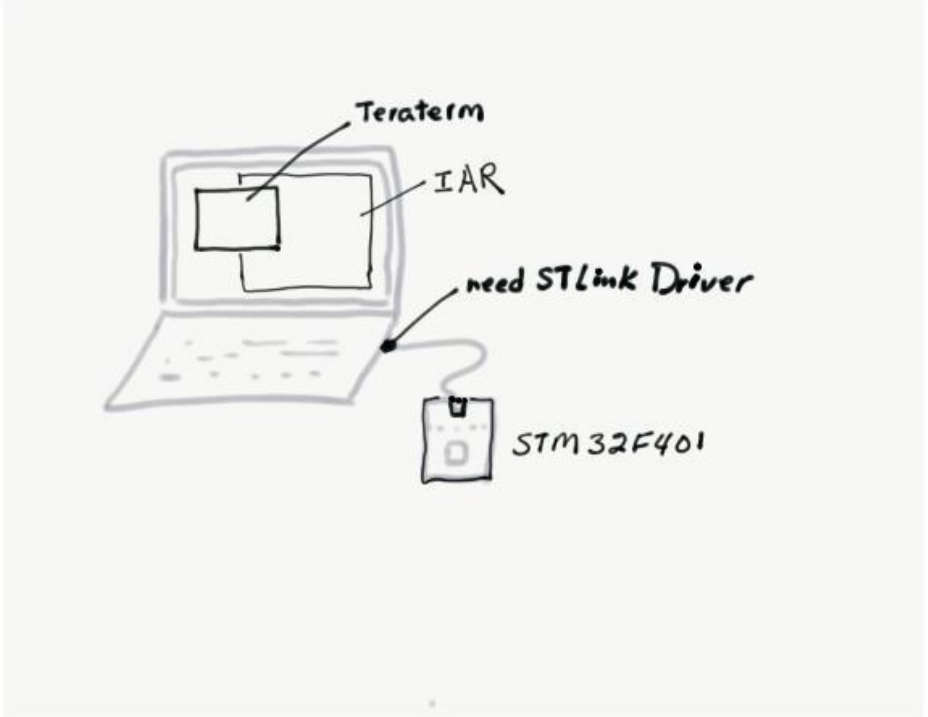


Running Program on STM32F401xE Nucleo

- IAR
- Teraterm

  A terminal of the board runs on Windows
- STLink

  An in-circuit debugger and programmer
- STM32F401



IAR – Settings for Simulator vs. Evaluation Board

RUNNING ON SIMULATOR SETTING:

1. Project → options → general options
   [Target] Device: ***STM32F32401RE***
2. C/C++ compiler
   Optimization: ***None***
3. Debugger → **Setup**
   Driver: ***Simulator***

RUNNING ON EVALUATION BOARD SETTINGS:

1. Precondition: install **STLink** driver, **Teraterm**
   Teraterm settings: **Serial** Port: COM3: STMicroslectornics STLink
2. C/C++ compiler
   Optimization: **None**
3. Debugger → **ST_LINK**
   Driver: **ST_LINK**
4. Download → use **flash loader**

C Programming

C & Embedded

- Why C?
- C Language History
- C in Embedded

C Environment

- File Structure
- Preprocessor
- Conditional Compilation

C Programming

- Variables
- Keywords
- Operators
- Flow control
- Demo 02 – Flow control
- Pointers
- Data Types
- Demo 03 – Variables & Pointers

C & Embedded

- Why C?
- C Language History
- C in Embedded

Why C?

- "High enough": Support function and modules
- "Low enough": Access hardware via pointers

- **Efficient**: size, speed, and memory usage
- **Easy** to understand, easy to use
- **Portability:** The same program can be compiled for different processors.

- Is C still popular?
  - https://www.tiobe.com/tiobe-index/

C Language History

- C was originally developed by Dennis Ritchie between 1969 and 1973 at Bell Labs, and used to re-implement the Unix operating system (B language was used for Unix OS prior to that).
- In 1978, Brian Kernighan and Dennis Ritchie published the first edition of "*The C Programming Language*". This became one of the most successful computer science books of all time.
- The rapid expansion of C over various types of hardware platforms led to many variations that were similar but often incompatible.
- This was a serious problem for portability, so a standard version of C was needed.
- And **C89** was born.
- Subsequent standards were created (**C99**, C11, & C18)

C in Embedded

C Programming	Embedded C Programming
Used for desktop computers	Used for Microcontroller based applications
C can use the resources of a desktop PC like memory, OS, etc.	Use the limited resources, such as RAM, ROM, I/O on an embedded processor.
Compilers for C typically generate OS dependent executables (.exe, .so)	Requires compilers to create files to be downloaded to the microcontrollers/microprocessors where it needs to run.

C Environment

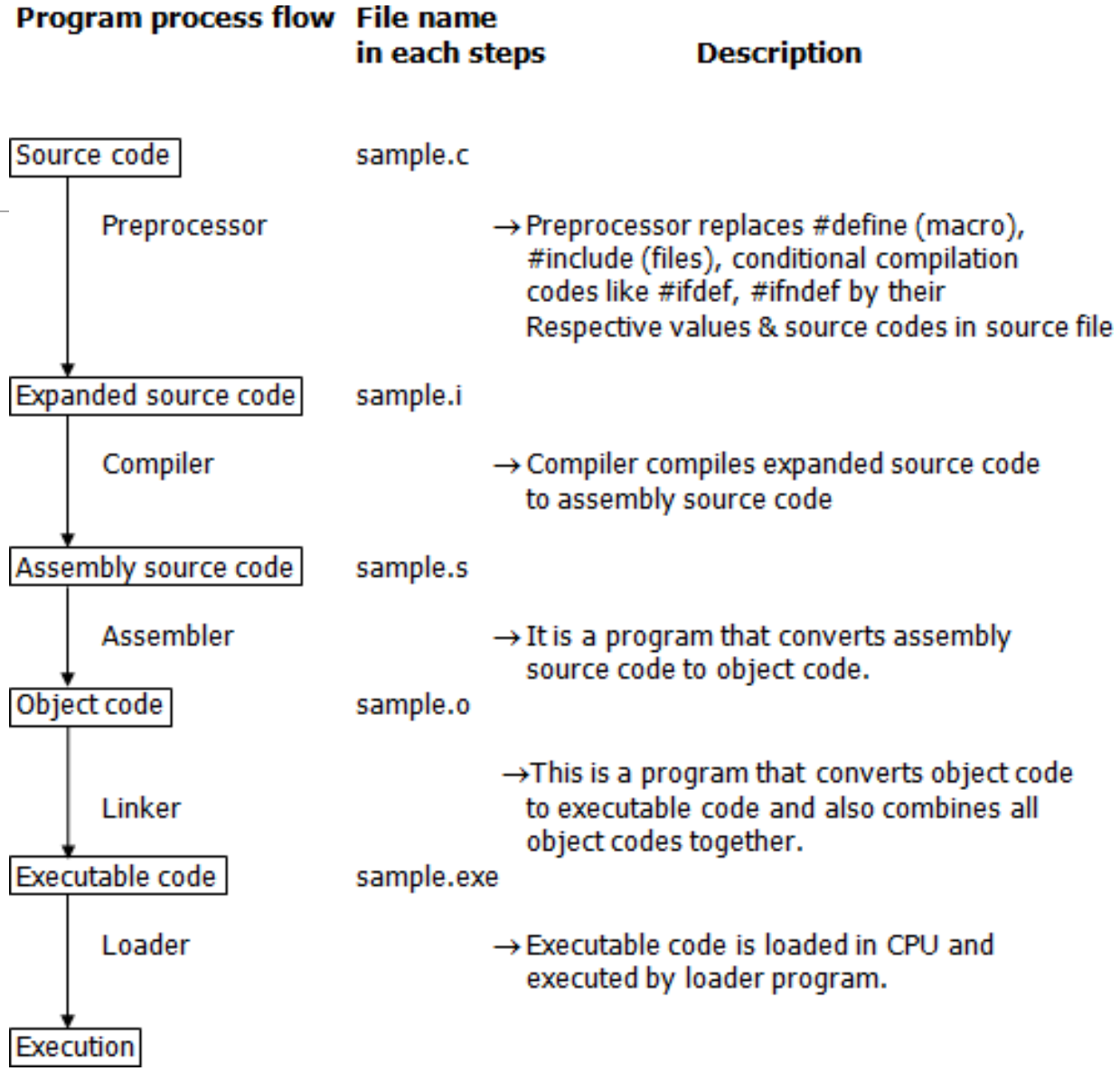
- C Files
- Preprocessor directives
- Conditional Compilation

C Files

- C programs generally consist of two types of files
  - Header (*.h) files
  - Implementation (*.c) files
- Header Files declare global (public)
  - Macros
  - Variables
  - Constants
  - Functions
- Implementation files (c files)
  - Define function implementations
  - Declare and define local (private) functions
  - Declare and define local (private) variables, constants, etc.

C Preprocessor

- Is part of the compiler
- Provides “text” processing of the C program files before they are actually compiled.



C Preprocessor - Directives

Directives are special instructions directed to the preprocessor

Directive	Function
#define	Defines a **Macro** Substitution
#undef	Undefines a **Macro**
#indude	Includes a File in the Source Program
#ifdef	Tests for a **Macro** Definition
#endif	Specifies the end of #if
#ifndef	Checks whether a **Macro** is defined or not
#if	Checks a Compile Time Condition
#else	Specifies alternatives when #if Test Fails

Preprocessor & Macros

The C preprocessor adds flexibility to software
— **macros:** text substitution strings
— **directives**: control compiler or preprocessor actions

directives
— Conditional compilation using #if, #ifdef, #ifndef, etc
— Copying contents of one file into another via #include
— Compiler customization via #pragma

macros
— Apply a label to a literal value to make code more reader-friendly and easier to maintain
— Define small inline function-like statements
— Define a control variable for, say, compiling different blocks of code depending on a value
— Encode bit values into a mask or control code

Preprocessor & Macros Example

```
// define masks for bits. Don't put comments at the end of the line because they can
// become part of the text expansion, which can make for tough to find bugs
#define BIT_0 (1<<0)
#define BIT_1 (1<<1)
#define BIT_2 (1<<2)

// define the address of a register. By convention, append reg addresses with _REG
#define CONTROL_REG 0x12345678

// define a few flags in the control register. By convention, append flag defines
// with _FLAG or _BIT
#define CONTROL_CLR_FLAG BIT_0
#define CONTROL_SET_FLAG BIT_1
#define CONTROL_ANY_FLAG BIT_2

// passing into a function to modify the register
setControlRegister( CONTROL_REG, CONTROL_CLR_FLAG );
```

Conditional Compilation

- Sometimes during development you may want to omit blocks of code from the build
  The usual way to do this is to "comment out" code
- A better way is to use the preprocessor

```
// You can use macros to conditionally compile code. Comment out the #define to omit
// the code in the block. This control flag (UART_INCLUDED) could be defined in the
// project, to create different build configurations
#define UART_INCLUDED 1

#ifdef UART_INCLUDED
   // then include the code in this block in the build
#endif


// absolute control example:
// #if interprets 0 as false and >0 as true. So "#if 0" always evaluates as false,
// so the code between it and its matching #endif will be omitted from the build
#if 0
   // ... some code
#endif
```

C Programming

- **Variables**
- **Keywords**
- **Operators**
- **Flow control**
- **Demo 02 – Flow control**
- **Pointers**
- **Data Types**
- **Demo 03 – Variables & Pointers**

Variables

- C is a case sensitive programming language
- Variables must follow these rules:
  - Cannot begin with a digit
  - Must consist of letters, digits or underscores
  - Cannot use a reserved key word (like “if”, “while”, “int”,…etc.)

Keywords

```
auto        enum        restrict    unsigned
break       extern      return      void
case        float       short       volatile
char        for         signed      while
const       goto        sizeof      _Bool
continue    if          static      _Complex
default     inline      struct      _Imaginary
do          int         switch
double      long        typedef
else        register    union
```

Reference: C99 Spec

C Operators

Operator	Description	Associativity
() [ ] . -> ++ --	Parentheses: grouping or function call Brackets (array subscript) Member selection via object name Member selection via pointer Postfix increment/decrement	left-to-right
++ -- + - ! ~ (*type*) * & sizeof	Prefix increment/decrement Unary plus/minus Logical negation/bitwise complement Cast (convert value to temporary value of *type*) Dereference Address (of operand) Determine size in bytes on this implementation	right-to-left
* / %	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <= > >=	Relational less than/less than or equal to Relational greater than/greater than or equal to	left-to-right
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
|	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
||	Logical OR	left-to-right
? :	Ternary conditional	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Addition/subtraction assignment Multiplication/division assignment Modulus/bitwise AND assignment Bitwise exclusive/inclusive OR assignment Bitwise shift left/right assignment	right-to-left
,	Comma (separate expressions)	left-to-right

WATCH!

Demo

0XA **&** 0XA ==> ??

0XA **&&** 0XA ==> ??

Flow control (if statement)

```
if (<condition_1>)
{
  <statement_1>;
}
else if (<condidtion_2>)
{
  <statement_2>;
}
else
{
  <statement_3>;
}
```

Flow control (switch-case statement)

SYNTAX:

```
switch (<controlling_expression>)
{
case <label_1>:
  <statement_1>;
  break;
case <label_2>:
  <statement_2>;
  break;
default:
  <statement_3>;
}
```

- The controlling expression of a **switch** statement can only have integer type.
- There may be at most one **default** label in a **switch** statement.
- **Watch out!**
  - Omission of the **break** statement causes execution to "***fall-through***" into the next case statement.

```
int watts;
int life;

switch (watts)
{
case 25:
  life = 2500;
  break;
case 40:
case 60:
  life = 1000;
  break;
default:
  life = 0;
}
```

Flow control (Loops)

WHILE LOOP & DO-WHILE LOOP

```
while (<condition is true>)
{
  //statement
}
-----------------------------------------------
do
{
  //statement
} while (<condition is true>);
```

FOR LOOP

```
for(<init_expression>;
<loop_condition>;
<update_expression>)
{
  //statement
}
```

Demo – 02: Flow Control

- Load counter example for a linear control flow
- Observe PC register
- Branch instruction
- CMP instruction (operates on the difference between two values)
- Note
  - Side effect to APSR
  - A C statement can translate into multiple Assembly statements
  - Compiler will optimize for best performance

BREAK 2

Pointers

- Pointers are considered a challenging aspect of C programming
- Pointers contain an address, nothing more

- C syntax
  - **&** “address of” operator is used to get the address of the variable.
  - ***** “dereference/value-at” operator is used to get the value of the variable that the pointer is pointing to.

Pointers illustrated

We can use the pointer to retrieve data from memory or to store data into memory

```
int myInt = 25;
int myOtherInt = 100;
int *myPointer;                 //Pointer declaration
myPointer = &myInt;             //Assigning address of myInt to myPointer
*myPointer = 50;                //Modify the value that myPointer is pointing to
myOtherInt = *myPointer  //Dereferencing == getting the value that is stored in the memory location
                                    pointed by myPointer. So myOhterInt will become  50/
```

Demo

C Data Types

C has two primary classes of data types

- Primitive (fundamental)
- Derived

Primitive types

- Supported by the typical machine hardware
- Integer types: char, short int, int, long int, both signed and unsigned
- Floating point types: float (single precision) and double (double precision)

Derived types

- Built upon the primitive types
- **Arrays** are groups of objects of the same type stored contiguously in memory
- **Functions** return an object of a given type (procedures, also called void functions, do not return an object)
- **Pointers** point to objects of a given type
- **Structures** contain groups of objects of any type
- **Unions** group objects of dissimilar type into one memory location

C Data Types Example

```
// Integar Types:
unsigned int A;          // unsigned integer
int B;                   // signed integer
unsigned char C;         // unsigned character
char D;                  // signed character

int F[256];              // An array of 256 signed integers
int *pa;                 // A pointer to a integer

int myFunct(void);       // A function that takes no arguments and returns an int

struct myStruct {        // A structure containing two integer type members
  int a;                 //   - each member stores a value
  char b;                //   - storage is allocated to hold all members
};

union {                  // A union holds one data member, it's type determined
  int a;                 // by how it's accessed. Enough storage is allocated to
  char b;                // hold the largest member
} u_name;
```

C Data Type Notes

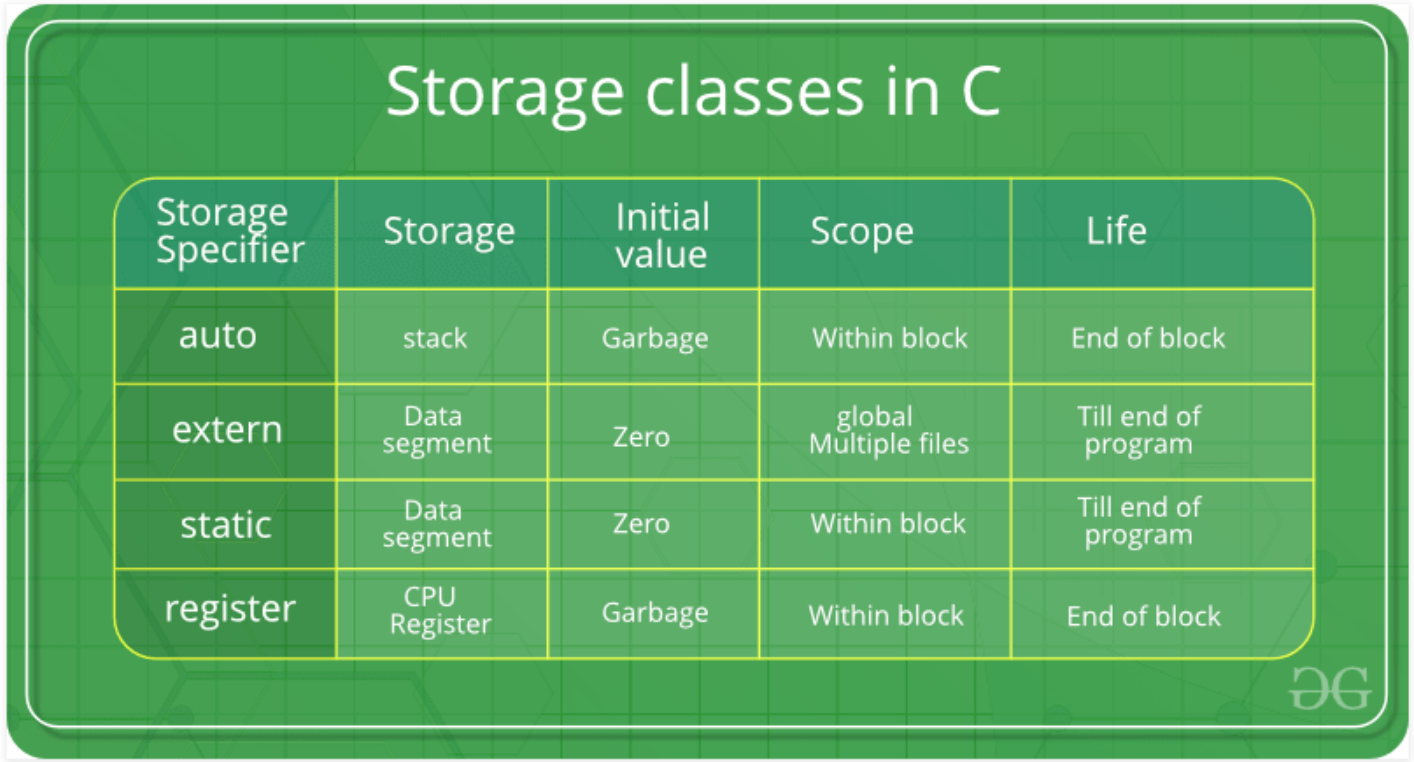
- Floating-Point Types
  - Very expensive software operation
  - Usually implemented in hardware in a floating-point unit (FPU)
  - Avoid using them in systems not having an FPU
- Integer types
  - Standard C integer types are machine-dependent
  - Example: an int reflects the size of the data word of the machine (8, 16, 32, 64 or 128 bits)
  - Hard to rely on size of variables
  - C now includes the header stdint.h, which defines integer types of a particular size
  - Example: uint32_t is an unsigned 32-bit integer, no matter what machine it's run on
  - Using the types in stdint.h increases portability

C Storage Specifiers & Qualifiers

- C gives the programmer control over storage and visibility of objects
- Storage Specifiers tell the compiler where to store the object
  - **auto**: store on the stack. Default for function local objects. Minimize these!
  - **static**: store in RAM
    - Maintain values between calls for function local objects
    - Limit visibility (scope) to the file if declared outside a function (unlike a global variable)
  - **register**: store in a register.
    - Ignored by most compilers except as hint that object could be frequently accessed.
  - **extern**: object declared externally (in another file or later in the same file)
  - **typedef**: renames a type
    - not really a storage specifier, but usually lumped in
- Type Qualifiers place limitation or conditions on a type
  - **const**: constant
    - Stored in Flash or ROM
    - The object can be initialized but not subsequently changed
  - **volatile**: suppress optimization. Probably the most important keyword in embedded C (we'll visit soon)

Storage Classes Summary

C language uses 4 storage classes, namely:



Source: https://www.geeksforgeeks.org/storage-classes-in-c/

Demo – 03: Variables and Pointers

- Making “counter” global
- Observe “counter”
- Load and Store machine instructions (LDR & STR)
- Convert program to use pointers

Assignment 02

Suggested Reading

- ***"An Embedded Software Primer" by David E. Simon***
  - Chapter 9 & 10
- ***"The Definitive Guide to ARM Cortex M3 & M4" by Joseph Yiu (Third Edition)***
  - Chapter 16
- ***"The C Programming Language" By Brian Kernighan & Dennis Ritchie (Second Edition)***
  - Chapters: 1, 2, 3, 4, 5, 6