Problems with Low-level Languages

- Programs in low-level languages require detailed documentation, as otherwise they hard to read for people who were not involved in the process of the program creation
- Example: Company "Ostrich" has recently re-developed their embedded software for flagship products
  - Developed in assembly, 80 percent working, 2000 lines of code
- Suddenly it has been realized that the product is not shippable
- Bugs: system lock-ups indicative of major design flaws or implementation errors + major product performance issues
- Designer has left the company and provided few notes or comments
- You are hired as a consultant. Do you:
  - Fix existing code?
  - Perform complete software redesign and implementation? In this case, which language?

Problem-oriented Language layer

- Compiled to assembly or instruction set level
- You will be using embedded C
- How does this differ from usual use of C?
  - Directly write to registers to control the operation of the processor
  - All of the registers have been mapped to macros
  - Important bit combinations have macros – use these, please !
  - Registers are 32 bits, so int type is 4 bytes
  - Register values may change without your specific instructions
  - Limited output system
  - Floating point operations very inefficient, divide + square-root to be avoided

Embedded C (more on MyCourses)

- C preprocessor
  - #define, #ifndef, #if, #ifdef, #else …etc.
    - #define specifies flags for conditional compilation
    - All remaining preprocessor statements initiate conditional compilation
      - Example: #ifdef compiles a block of code if some condition is defined in the #define statement
  - #ifndef is used to prevent multiple includes
    - Use #ifndef if you want to include a new definition
  - Widely used in Firmware (embedded SW) drivers

C Preprocessor Examples

Inline macro functions:

```
#define MIN(n,m) (((n) < (m)) ? (n) : (m))

#define MAX(n,m) (((n) < (m)) ? (m) : (n))

#define ABS(n) ((n < 0) ? -(n) : (n))
```

Macro used to set LCD control (## provides the way to concatenate actual arguments during macro expansion)

```
#define SET_VAL(x) LCD_Settings.P##x
```

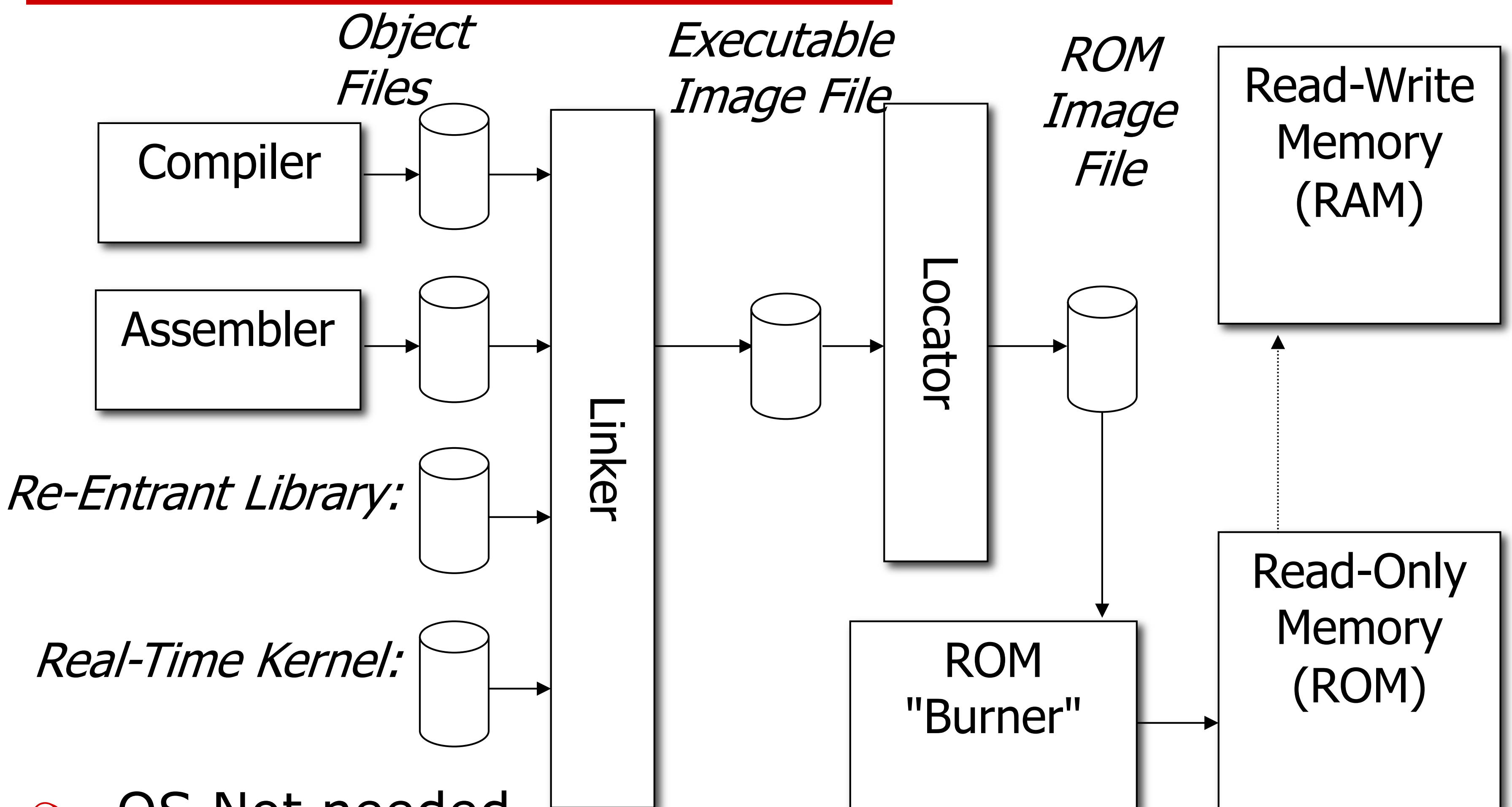
Nested macro definitions

```
#define SET(x, val) SET_VAL(x) = val
#define DEF_SET(x) SET(x, DS_P##x)
```

Global variables

- Distinguish global variables from local by choosing appropriate naming convention
  - Example: RX_Buffer_Gbl
  - Stick to your convention throughout the program
- Use them as Software flags
  - Example: PACKET_RECEIVED – use capitals
- Have them all in ONE place
- With your SW tool, global variables are easy to observe during debug (“watch variables”)

Embedded Build and Load



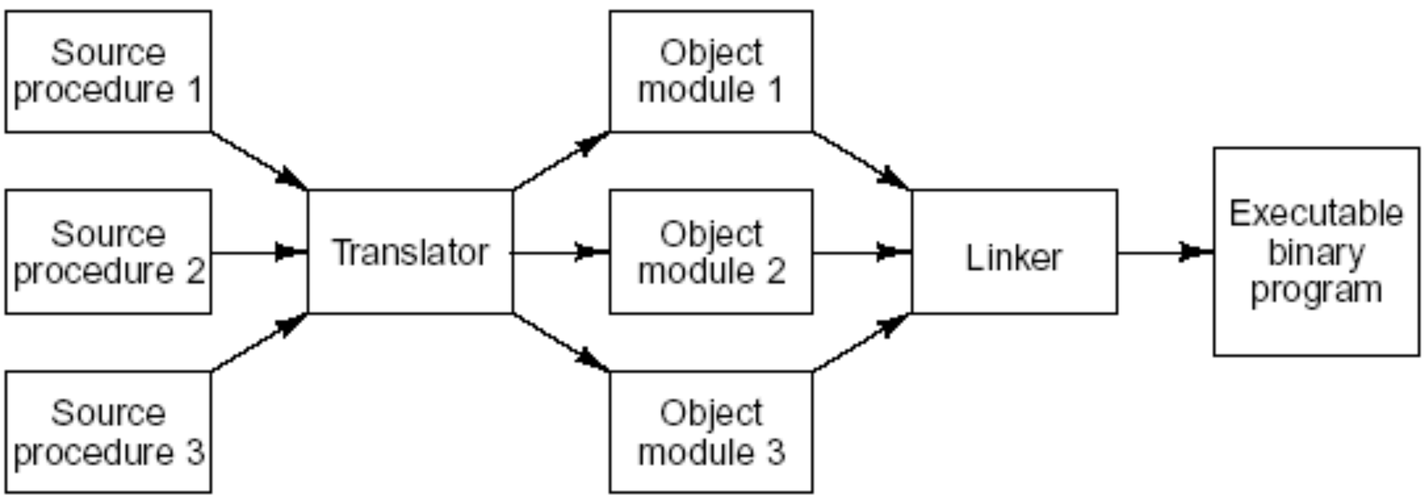
- OS Not needed

Linker

- Each procedure in assembly is translated individually and stored in the translated output disk
  - Once all the procedures are translated they have to be put together, i.e., linked, before the program can run
- When assembly source file is assembled by an assembler and C source file is compiled by C compiler, then the resulting two object files can be linked together by a software called **linker** to form the final executable
- Steps in program translation
  - Compilation (C programs) or assembly (assembly programs) of the source procedures
    - Task performed by compiler (assembler)
  - Linking of object modules
    - Done by linker

Modules - Basic Concepts

- Linking multiple files
  - Good SW design techniques
- Principle: have standardized modules
  - Some overhead – module structure

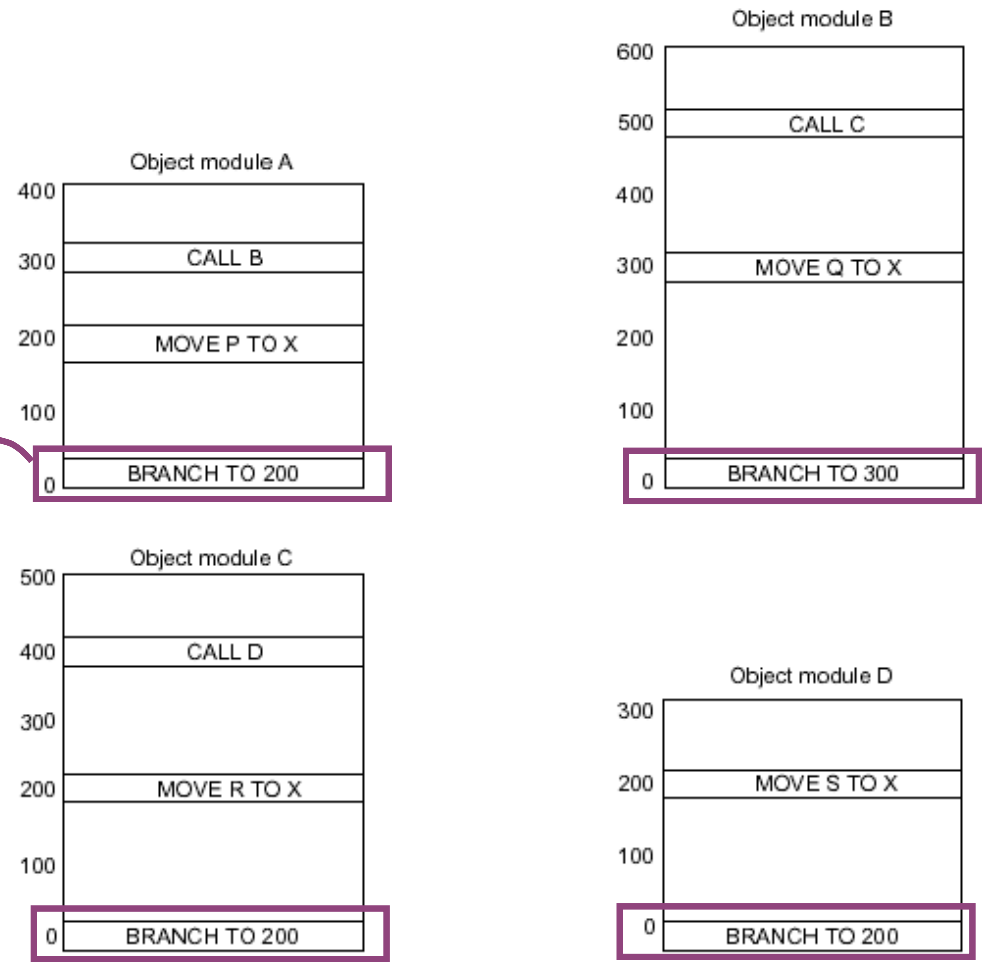


Benefits of Individual Procedure Compilation

- Assembly files can be written using any syntax and assembler available
- All the assembly code exists in a separate file, which is comfortable if any changes are required in the assembly code
  - If compiler/assembler were to translate a series of source procedures directly into a machine language, then any changes in the original source file would require retranslation of all the source file into machine language
  - When each procedure is translated into a separate file, then upon changes to the source file of this procedure it is necessary to retranslate only the source file of the procedure, while all the remaining code stays the same
    - Relinking of a retranslated file to the rest of the code is required

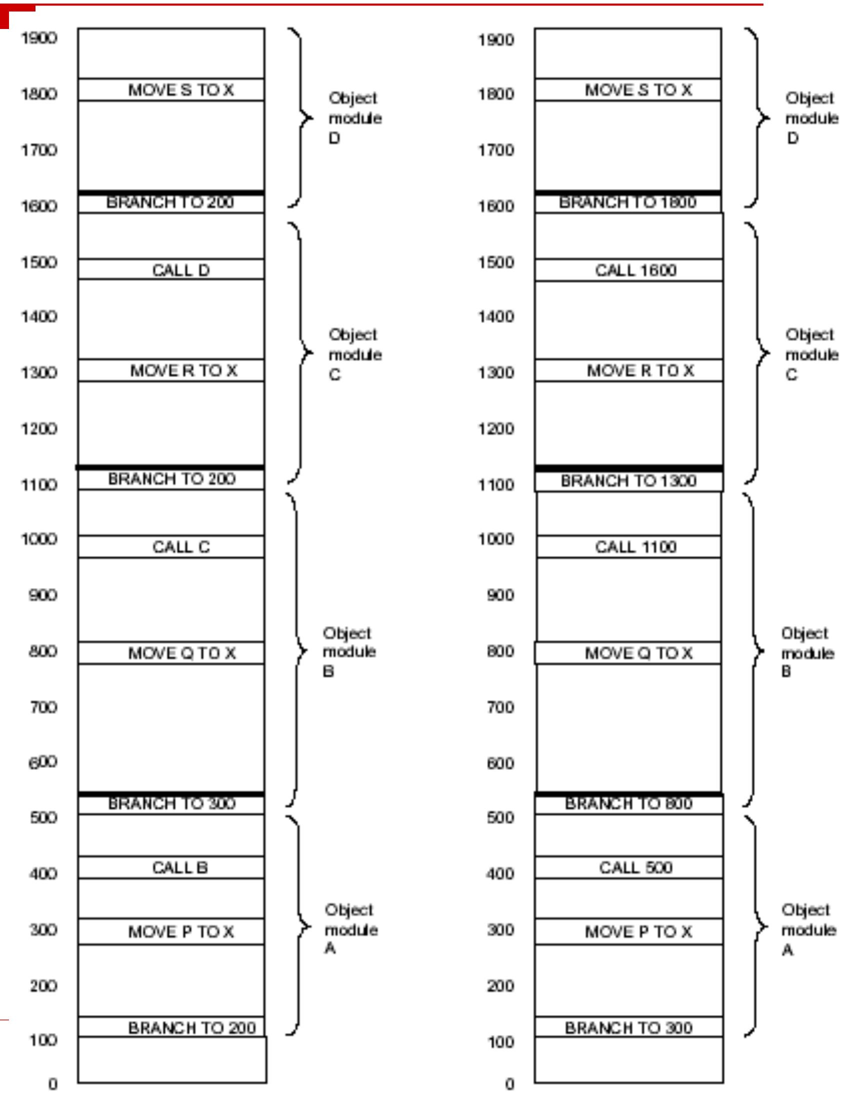
Module - Location in Virtual Memory

- At the beginning of assembly translation instruction counter is set to 0
  - Object module will be located at (virtual) address 0 during execution
- Example: 4 modules each beginning with BRANCH



Module "Composition"

- Linker brings all objects into main memory
  - By that the exact image of executable program's virtual address space inside the linker is formed
    - Typically few initial memory locations are reserved for various communications with operating system and is not accessible for program data
      - In the example the starting memory location is 100



Relocation Problem - Problems with Address Changes

- The created image of executable binary program has wrong reference addresses for branch instructions
  - Addresses were created for modules, whose starting instruction were all located at address 0 of virtual memory
    - That means that the branch addresses of each of the four modules in the example were calculated w.r.t. address 0 of the first object instruction
    - Upon changing the initial address of each of the object instructions, the addresses pointed by the branch instructions must be recalculated
- **Relocation problem** happens as each object module represents a separate address space

McGill

Solving Relocation Problem

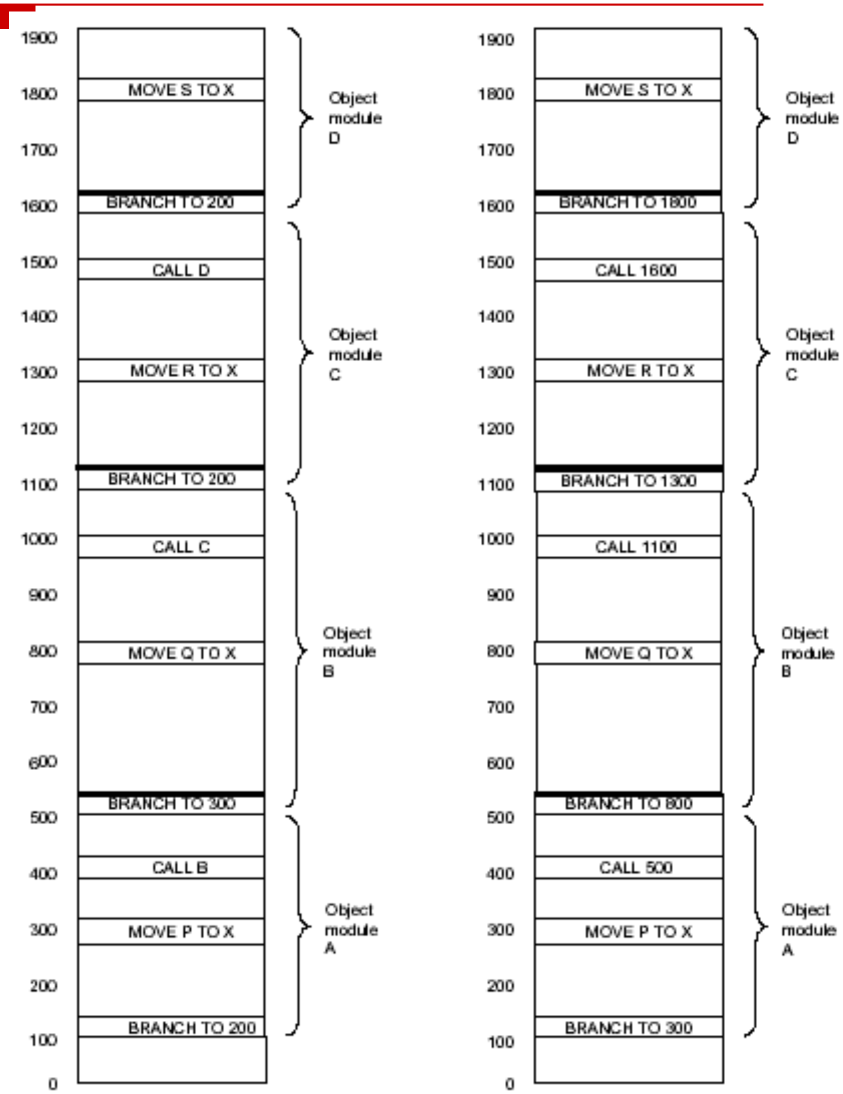
- Linker solves the relocation problem by:
  - Getting an exact estimate of the sizes of all object modules
  - Assigning the starting address to the first module
  - Assigning the right reference addresses for branch and jump procedures inside each module based on the starting address and the size of each module
- The actual steps taken by linker can be summarized as:
  - Construction of a table of all object modules and their length
  - Assignment of a starting address to each object module based on the above table
  - Addition of the relocation constant to all instructions, which reference memory
    - Relocation constant is equal to the starting address of its module
  - Finding of all instructions that reference other procedures and inserting the address of these procedures in place

Example: Object Module Table

- The object module table for the modules to the right

Module	Length	Starting Address
A	400	100
B	600	500
C	500	1100
D	300	1600

- Most linkers require two passes
  - During the first pass the linker reads all the object modules and builds up a table of module names and length, and a global symbol table consisting of all entry points and external references
  - During the second pass the object modules are read, relocated and linked one module at a time



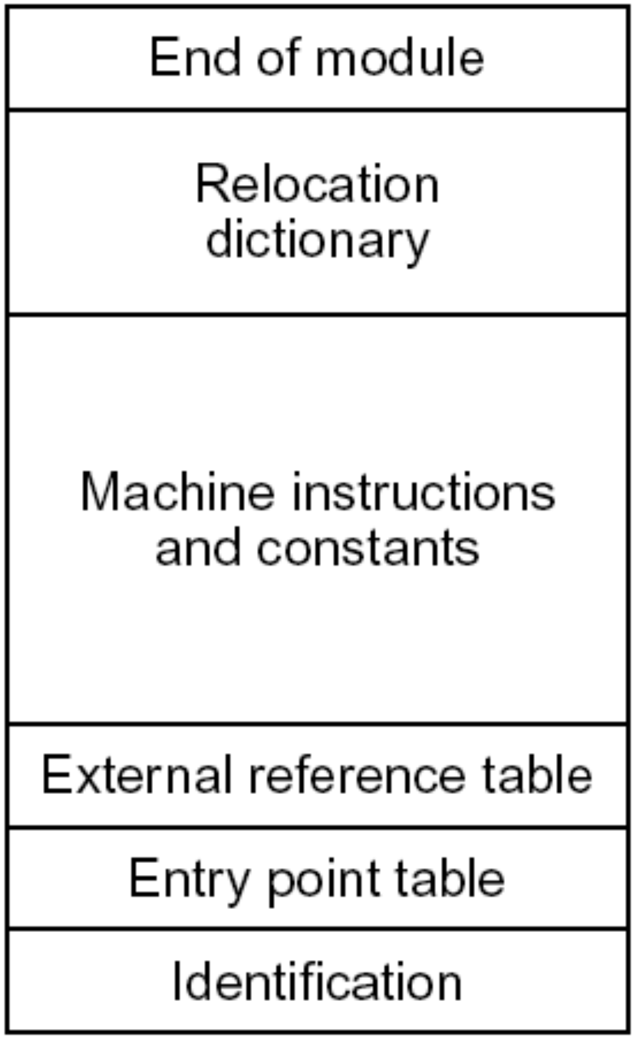
Object Modules

- Six different components
  - First part: name of the module and likner information such as length of various parts of module
  - Second part: list (and value) of symbols defined in this module subject to references by other modules
  - Third part: list of symbols that are used in the module but are defined in other modules (including list of all machine instructions used with each symbol)
  - Fourth part: assembled code and constants
    - The only part which is actually loaded into memory to be executed
    - Other parts will be used only by linker and be discarded before the execution begins
  - Fifth part: relocation dictionary
    - Instructions in each module which contain memory addresses must have a relocation constant added when put all modules are put together
  - Sixth part: end-of-module mark

End of module
Relocation dictionary
Machine instructions and constants
External reference table
Entry point table
Identification

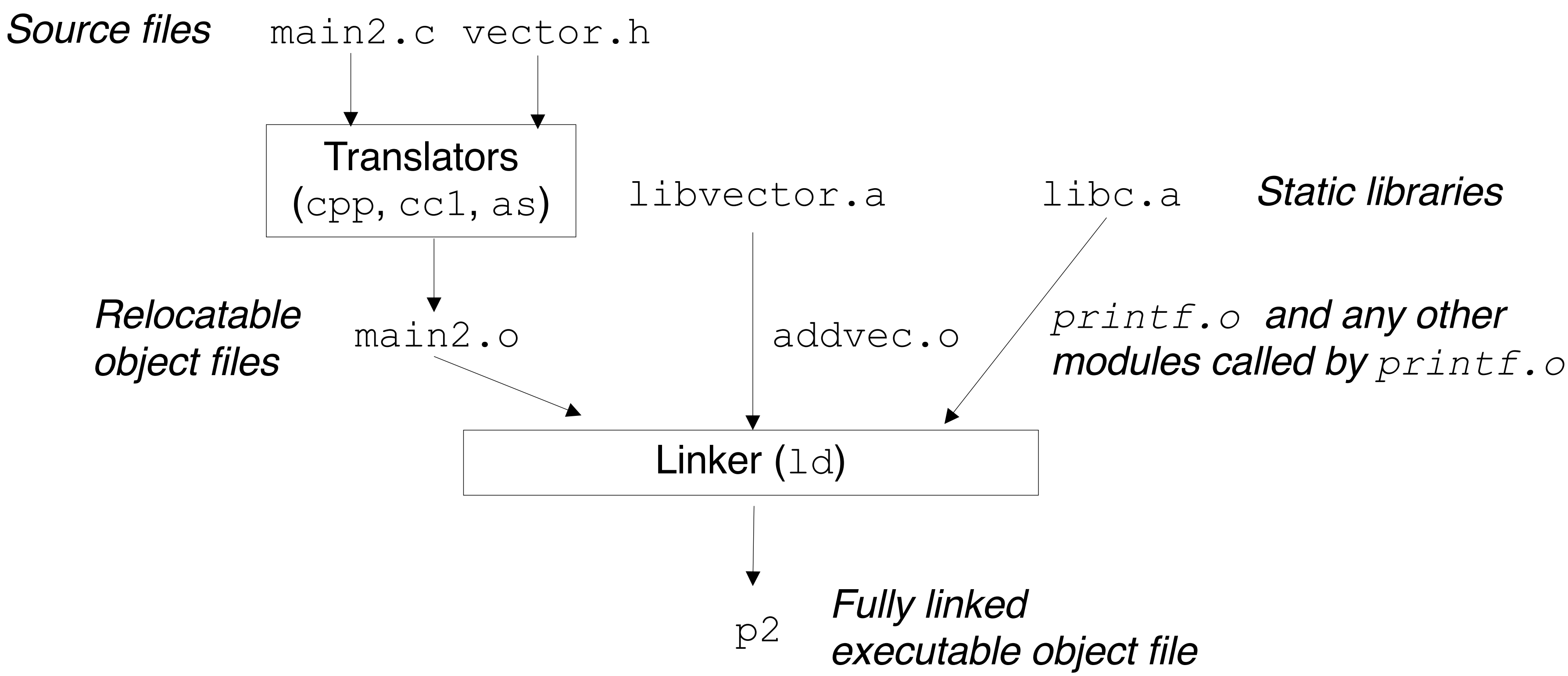
Object Modules

- Six different components
- Module standards
  - Unix
    - ELF
    - Shared Objects (.so files)
    - .o files
  - Windows
    - DLL
    - .obj



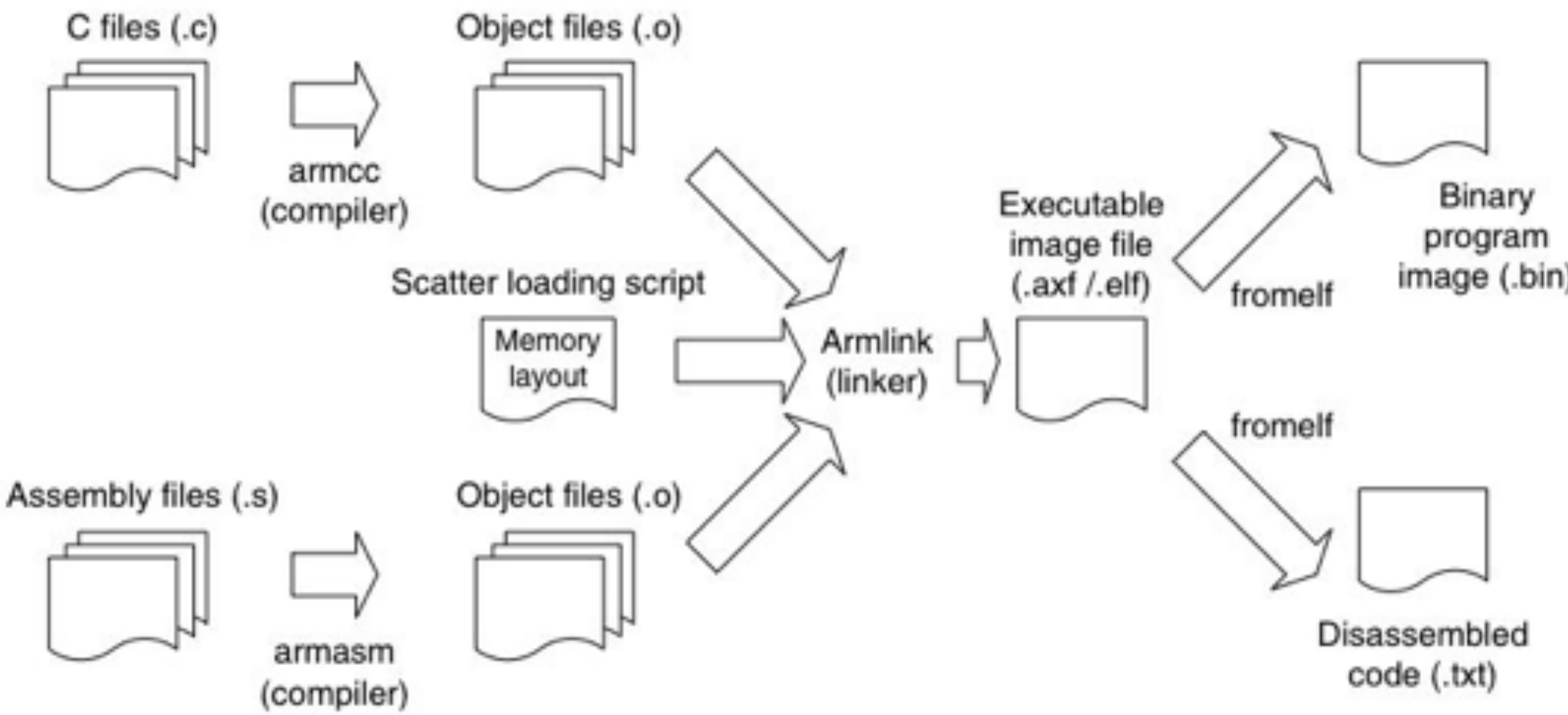
Linking Together

- Static Library - Unix/Linux



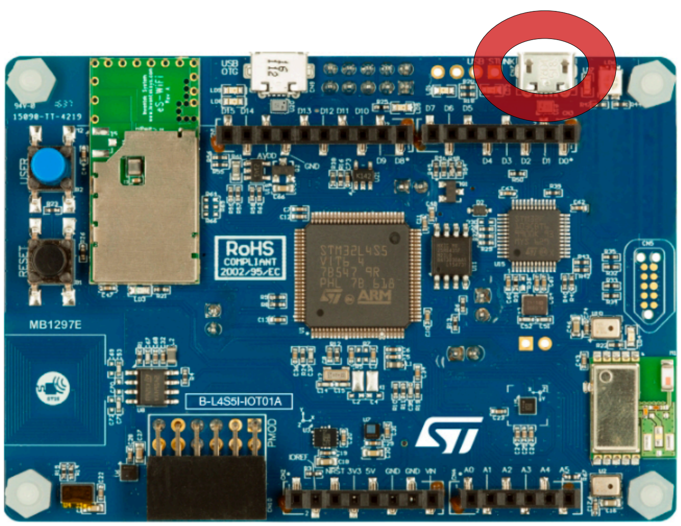
ARM Tools Flow: Recap

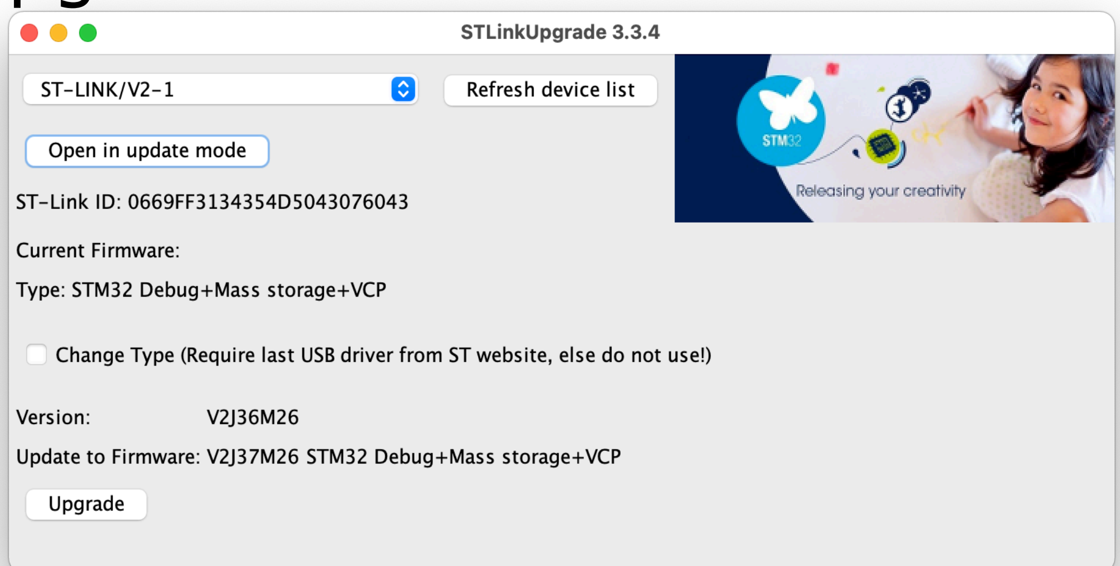
- ARM Standard using ARM Tools



STM32CubeIDE+Board Hints

- Integrated: single download and install
  - On Windows might need STLink tool, driver
- Checking STLink (connect the board!)
  - Help->ST-LINK Upgrade





FP Compilation Options

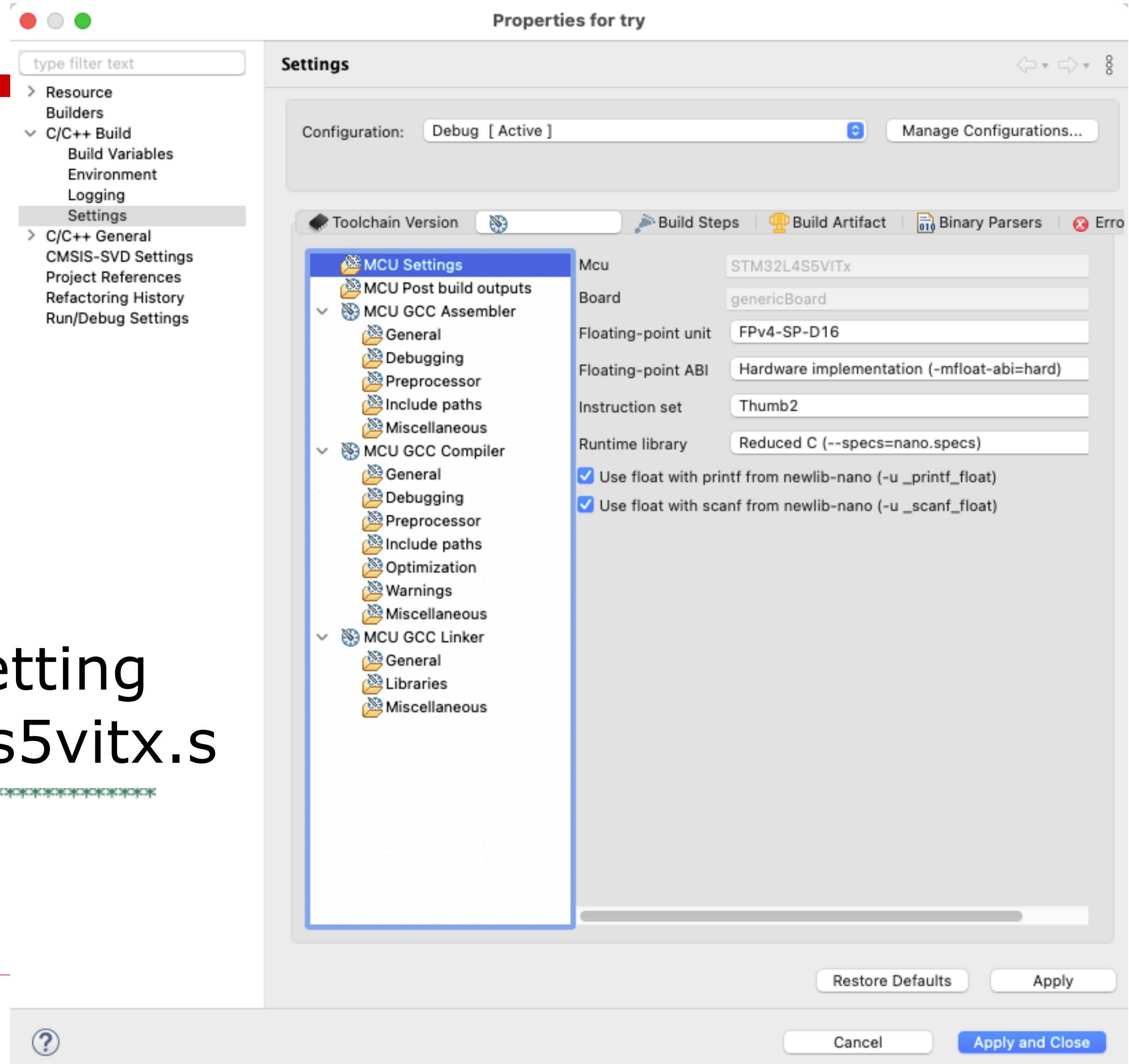
Access at:

Project->

Properties

(C/C++ Build

-> Settings)



Check assembler setting in startup_stm32l4s5vitx.s

```
****************************************
*/

.syntax unified
  .cpu cortex-m4
  //.fpu softvfp
  .thumb
```

Outline

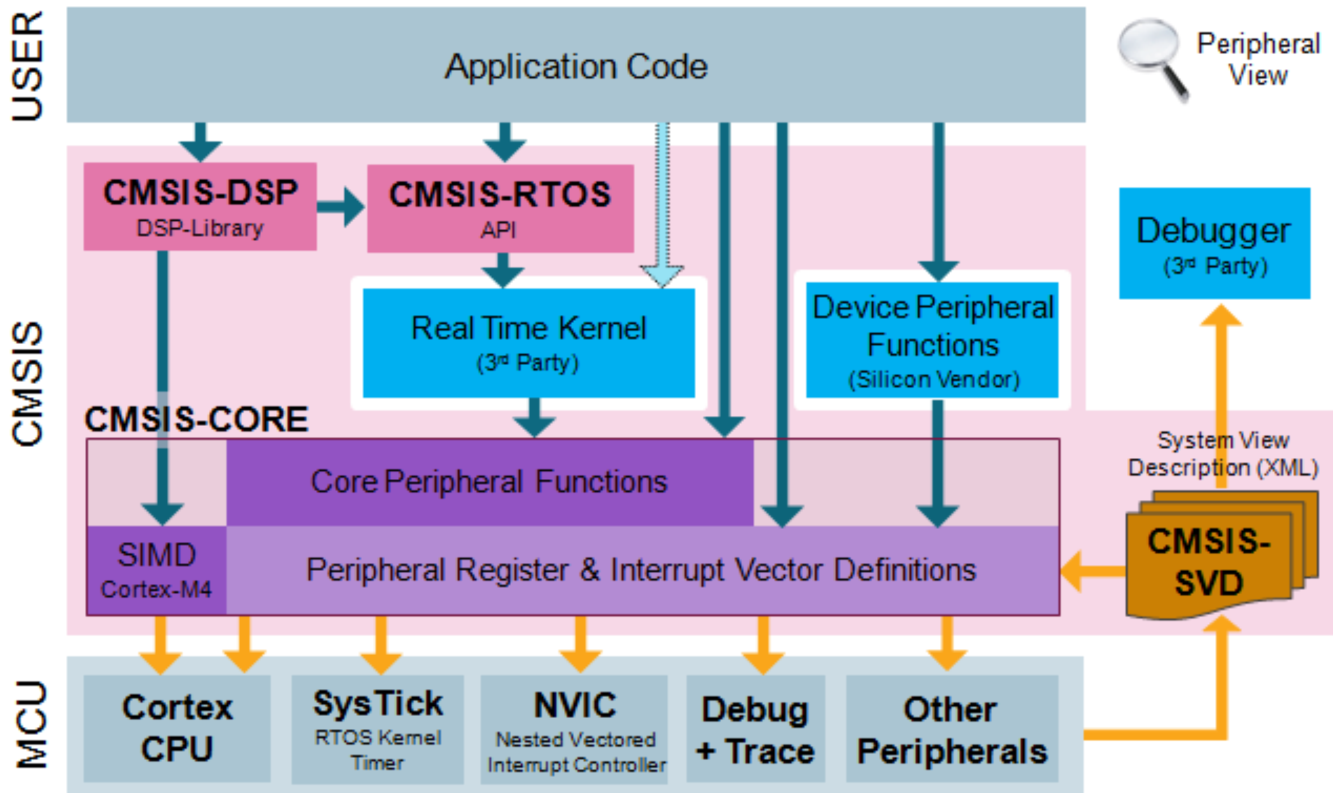
- CMSIS-DSP
- ARM Cortex M3 & M4 Families
- Practical Lab Issues
- Processor Microarchitecture
- In Tutorial/Lab: SW Infrastructure: CUBE, Q&A

Another Layer - CMSIS

- Added abstraction layer
- Hardware-independence
  - Across families
  - Implementation-independent
- SW Development enhancement
  - Speed of development
  - Quality & performance of code
- Imposes discipline in coding

CMSIS Overview - Simplified

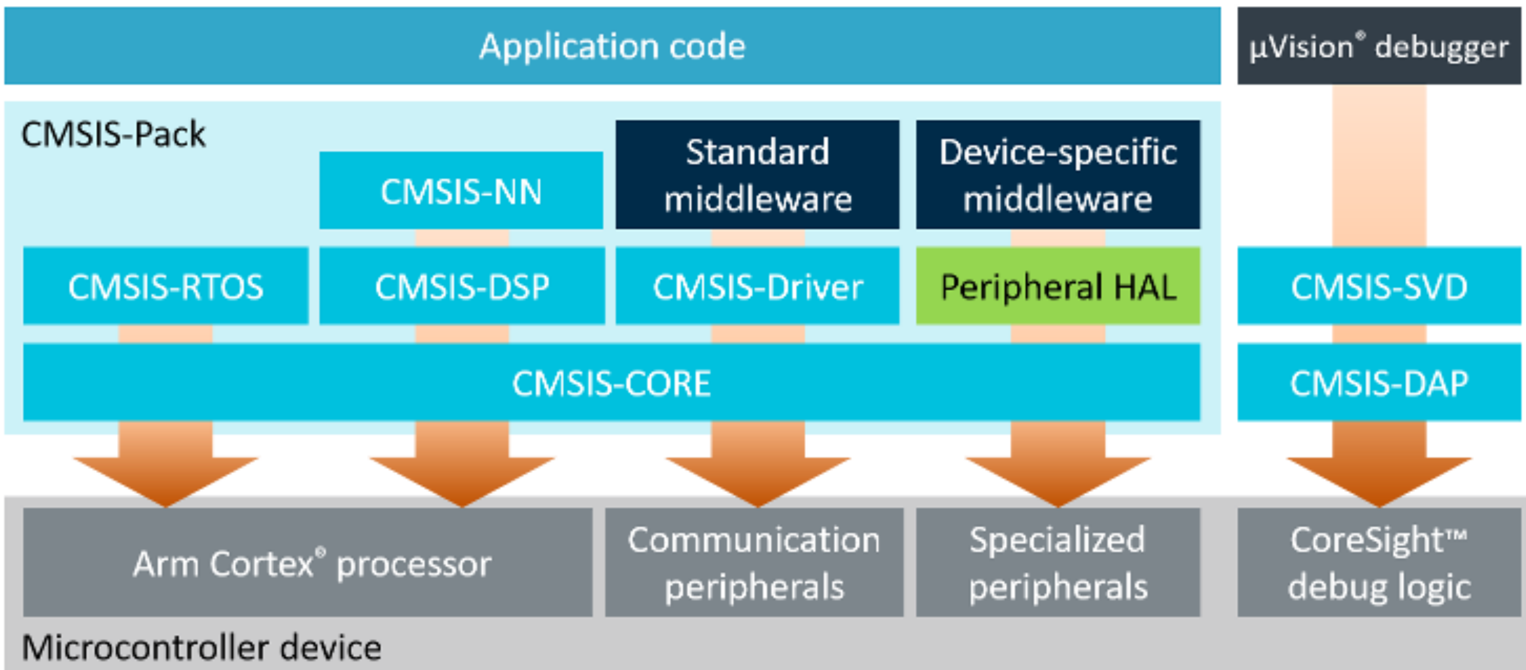
- Several parts, module dependencies



CMSIS Modules (APIs)

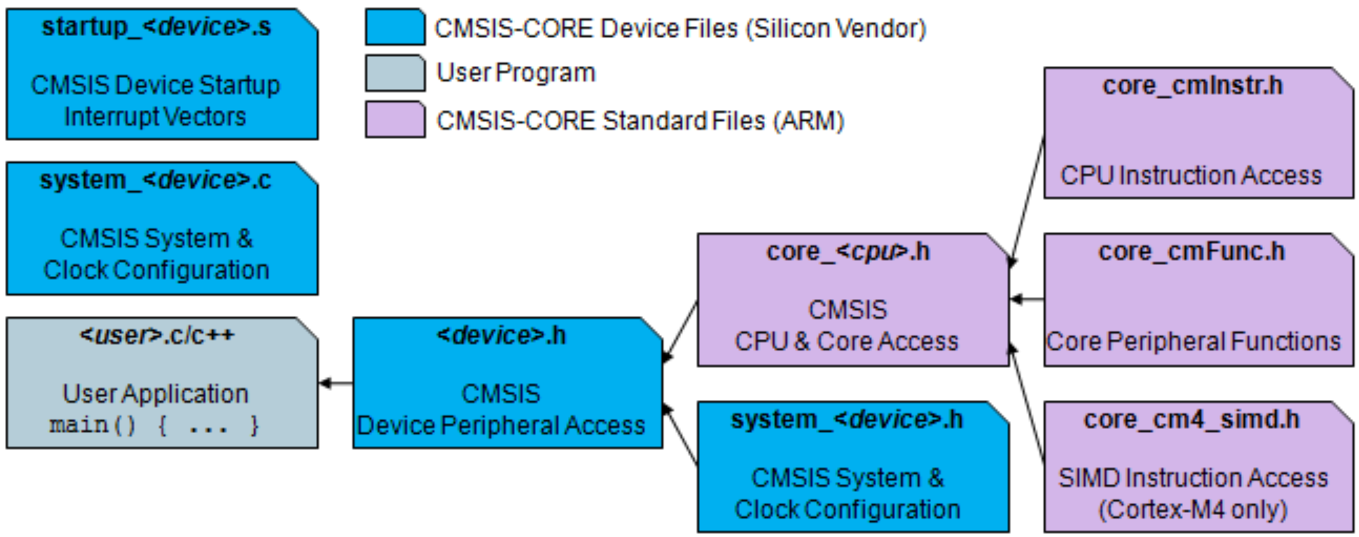
- **CMSIS-CORE**: Provides interface to Cortex-M0, Cortex-M3, Cortex-M4, … processors and peripheral registers
- **CMSIS-DSP**: DSP library with over 60 functions in fixed-point (fractional q7, q15, q31) and single precision floating-point (32-bit) implementation
- **CMSIS-RTOS**: Standardized programming interface for real-time operating systems for thread control, resource, and time management
- **CMSIS-SVD**: System View Description XML files that contain programmer's view of a complete microcontroller system including peripherals
- **CMSIS-DAP**: Debug access point
- **CMSIS-NN**: Neural Networks, rely on CMSIS-DSP

CMSIS-DSP as a Springboard



Setting-up CMSIS-CORE

- The CMSIS-CORE File structure consists of:
  1. CMSIS-CORE Device Files (from Silicon Vendor)
  2. CMSIS-CORE Standard Files (from ARM)
  3. User Files



McGill

CMSIS-DSP Function Summary

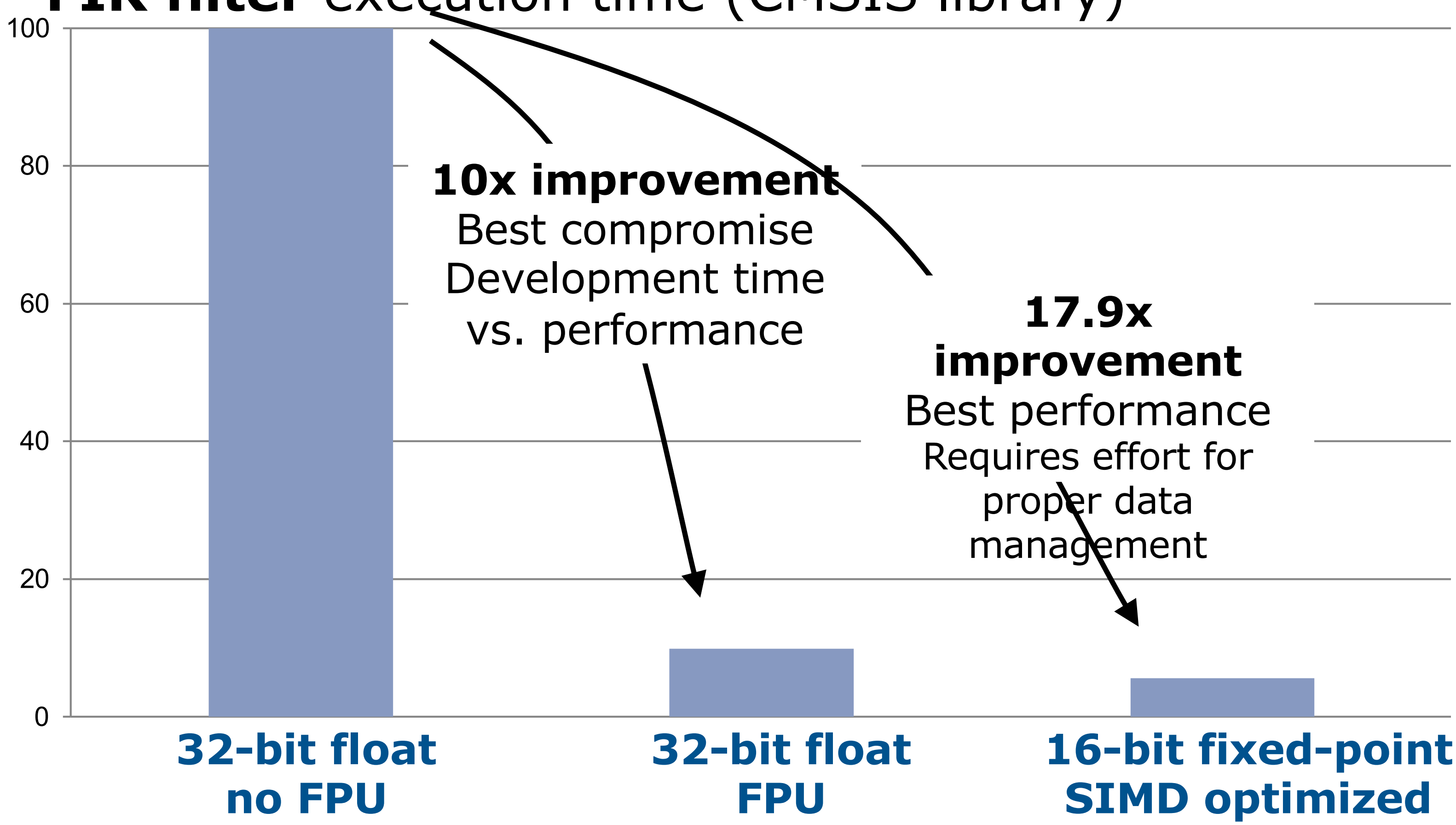
Library of functions implementing:

- Basic math functions
- Fast math functions
- Complex math functions
- Filters
- Matrix functions
- Transforms
- Motor control functions
- Statistical functions
- Interpolation functions
- Support functions

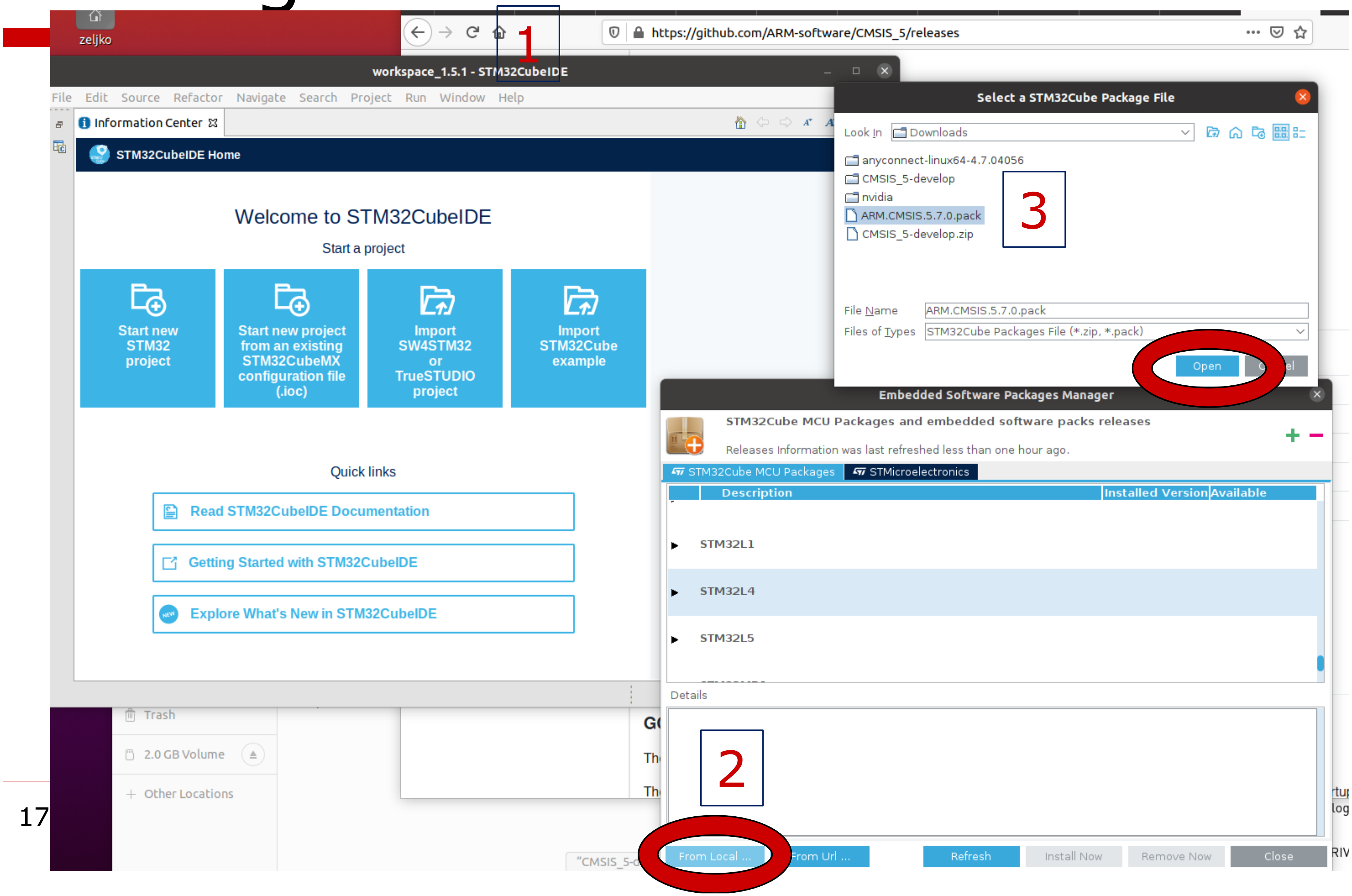
Documentation posted on MyCourses – html/online form

DSP Performance – Hardware Targets

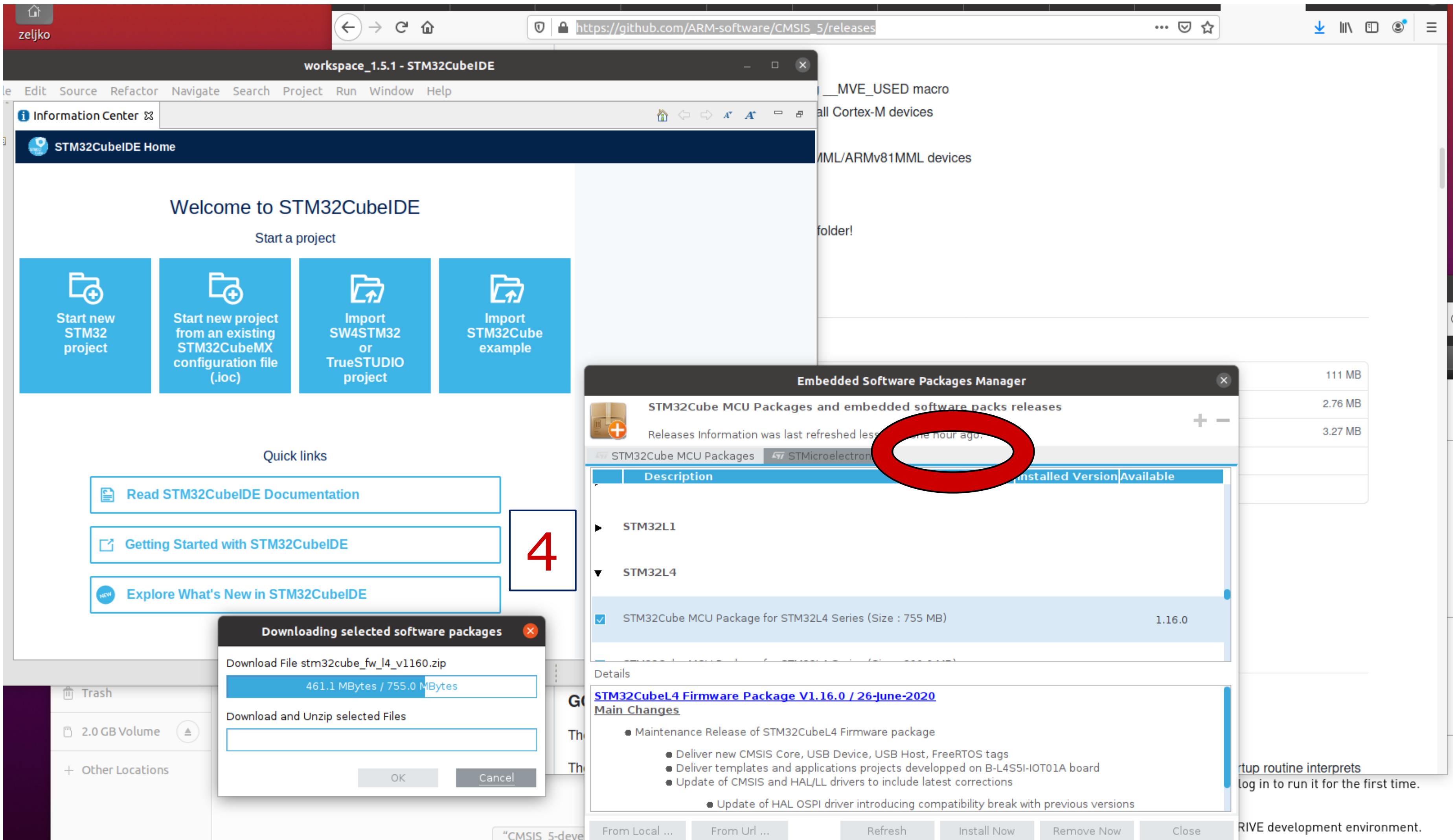
- **FIR filter** execution time (CMSIS library)



Getting CMSIS-DSP to CubeIDE



Getting CMSIS-DSP to CubeIDE



Some Practice: Conditional, IT

```
if (a > 10) {            cmp   r0, #10
  a = 10;                movgt r0, #10
} else {                 addle r0, r0, #1
  a = a + 1;
}
```

```
if (a > 10) {            cmp   r0, #10
  a = 10;                ite   le
} else {                 addle r0, r0, #1
  a = a + 1;             movgt r0, #10
}
```

Subroutines and Assembly Code

- Saving the context: examples

Main Program

```
    ...
 ; R4 = X, R5 = Y, R6 = Z
 BL     function1
```

Subroutine

```
function1
     PUSH    {R4-R6} ; Store R4, R5, R6 to stack
     ... ; Executing task (R4, R5 and R6
       ; could be changed)
     POP     {R4-R6} ; restore R4, R4, R6
     BX      LR    ; Return
```

```
; Back to main program
; R4 = X, R5 = Y, R6 = Z
... ; next instructions
```

Main Program

```
    ...
 ; R4 = X, R5 = Y, R6 = Z
 BL     function1
```

Subroutine

```
function1
     PUSH    {R4-R6, LR} ; Save registers
                 ; including link register
     ... ; Executing task (R4, R5 and R6
         ; could be changed)
     POP     {R4-R6, PC} ; Restore registers and
                 ; return
```

```
; Back to main program
; R4 = X, R5 = Y, R6 = Z
... ; next instructions
```

```
xyz:
    push  r4
    mul   r4, r1, r2
    add   r0, r0, r4
    pop   r4
    bx    lr

main:
    mov   r0, #1
    mov   r1, #2
    mov   r2, #3
    bl    xyz
    ldr   r1, =0x12345678
    str   r0, [r1]
```

[Source: Elsevier]

Embedded Processors

- Microcontrollers: Embedded Processors (8 to 64-bit)
  - Lots of peripherals added on chip
  - Single-chip operation modes and extendible modes
    - Flexible IOs
- Modern microcontrollers
  - Utility (for given application targets)
  - Cost
  - Low power operation (mobile, wireless, battery powered)
- Software and (Intellectual Property) IP core support
  - Increasingly important