CSE 3113: Microprocessor and Assembly Language Lab

Professor Upama Kabir

Computer Science and Engineering, University of Dhaka,

Lab 1

January 23, 2024

Outline

- Required Software Tools
- Why Keil MDK
- Software Development Layers
- Memory structure of Cortex-M
- Compiling
- Memory Layout

Required Software Tools

1. Install Keil for windows
   `https://www.keil.com/download/product/`
   Inside here choose MDK-Arm
2. Install ST-LINK debugger for windows
   `https://www.st.com/en/development-tools/stsw-link009.html`
3. Install STM CubeMX (Optional: only for better understanding as it gives a graphical representation but don't use it for your programming)
   `https://www.st.com/en/development-tools/stm32cubemx.html`

Why Keil MDK

Keil MDK (Microcontroller Development Kit) is the complete software development environment for a range of Arm Cortex-M based microcontroller devices. MDK includes:

1. μVision IDE with Integrated Debugger, Flash programmer and the Arm® Compiler toolchains.
2. STM32CubeMX exports μVision projects.
3. FreeRTOS, RTX and Micrium are directly supported
4. Keil Middleware: Network, USB, Flash File and Graphics.
5. Arm Compiler 5 and Arm Compiler 6 (LLVM) are included. GCC is supported.

Software Packs

- Software packs can be added anytime which makes new device support and middleware updates independent from the toolchain:
  - Device support is added via device family packs listed on the CMSIS-Pack index.
  - CMSIS offers software packs that contain components for core support, DSP and NN libraries, and a free-to-use real-time operating system.
  - MDK-Middleware provides royalty-free software components for communication peripherals in microcontrollers (TCP/IP, USB, file system, and graphics).
  - The Arm FuSa Run-Time System is a set of embedded software components qualified for use in the most safety-critical applications in automotive, railway, medical, and industrial systems.

Software Development Layers

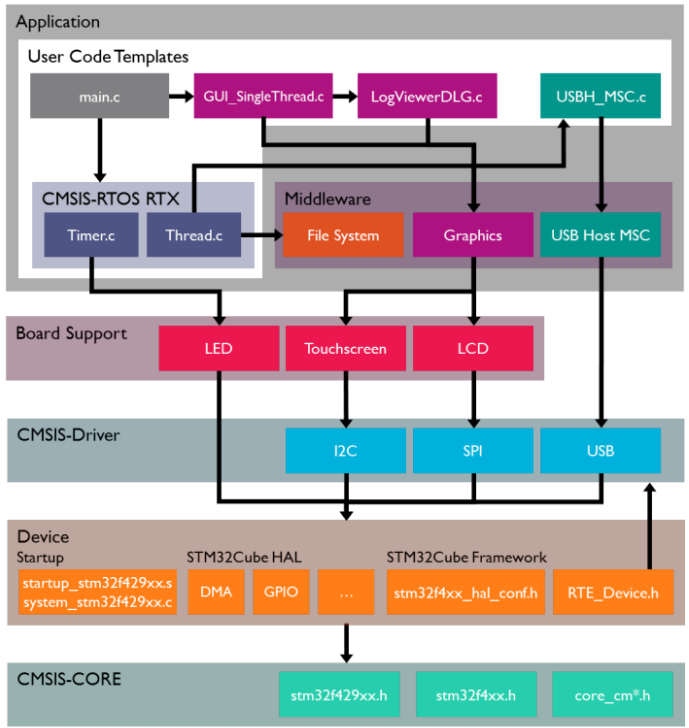


Figure 1

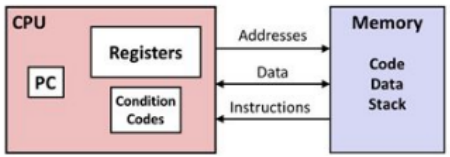
Levels of Abstraction

- C [and other high level languages] are easy for programmers to understand, but computers require lots of software to process them
- Machine code is just the opposite: easy for the computer to process, humans need lots of help to understand it
- Assembly language is a compromise between the two: readable by humans (barely), close correspondence to machine code

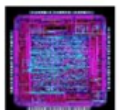
C programmer

```
#include <stdio.h>
int main(){
  int i, n = 10, t1 = 0, t2 = 1, nxt;
  for (i = 1; i <= n; ++i){
    printf("%d, ", t1);
    nxt = t1 + t2;
    t1 = t2;
    t2 = nxt; }
  return 0; }
```

Assembly programmer



Computer designer



Gates, clocks, circuit layout, ...

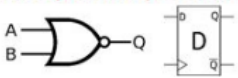


Figure 2

What does it mean to compile code?

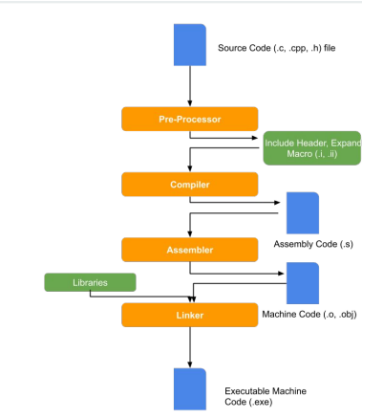


Figure 3

- Computer follows steps to translate your code into something the computer can understand
- This is the process of compiling code [a compiler completes these actions]
- Four steps: (i) preprocessing, (ii) compiling, (iii) assembling, (iv) linking

Pre-Processor

- Peculiar to the C family; other languages don't have this
- Processes #include, #define, #if, macros
  - Combines main source file with headers (textually)
  - Defines and expands macros (token-based shorthand)
  - Conditionally removes parts of the code (e.g. specialize for Linux, Mac, ...)
- Removes all comments
- Output looks like C still

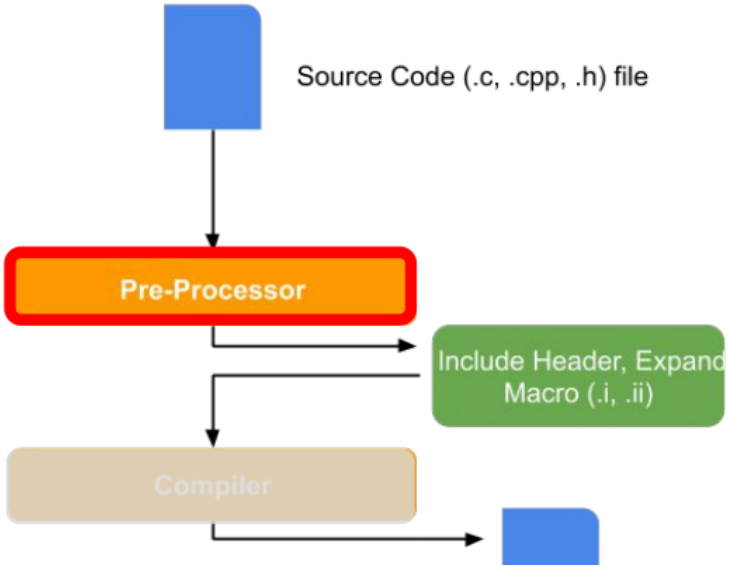


Figure 4

Before and after preprocessing

```c
#include <limits.h>
#include <stdio.h>

int main(void) {
    // Report the range of `char` on this system
    printf("CHAR_MIN = %d\n"
           "CHAR_MAX = %d\n",
           CHAR_MIN, CHAR_MAX);
    return 0;
}
```

- Contents of header files inserted inline
- Comments removed
- Macros expanded
- "Directive" lines (beginning with #) communicate things like original line numbers

```c
# 1 "test.c"
# 1 "/usr/lib/gcc/x86_64-linux-gnu/10/include/limits.h" 1 3 4
...
# 1 "/usr/include/stdio.h" 1 3 4
...
extern int fprintf (FILE *__restrict __stream,
        const char *__restrict __format, ...);
extern int printf (const char *__restrict __format, ...);
...
# 874 "/usr/include/stdio.h" 3 4
# 3 "test.c" 2

int main(void) {
    printf("CHAR_MIN = %d\n"
           "CHAR_MAX = %d\n",
# 6 "test.c" 3 4
           (-0x7f - 1)
# 6 "test.c"
                   , 0x7f);
    return 0;
}
```

Figure 5

C to Machine Code

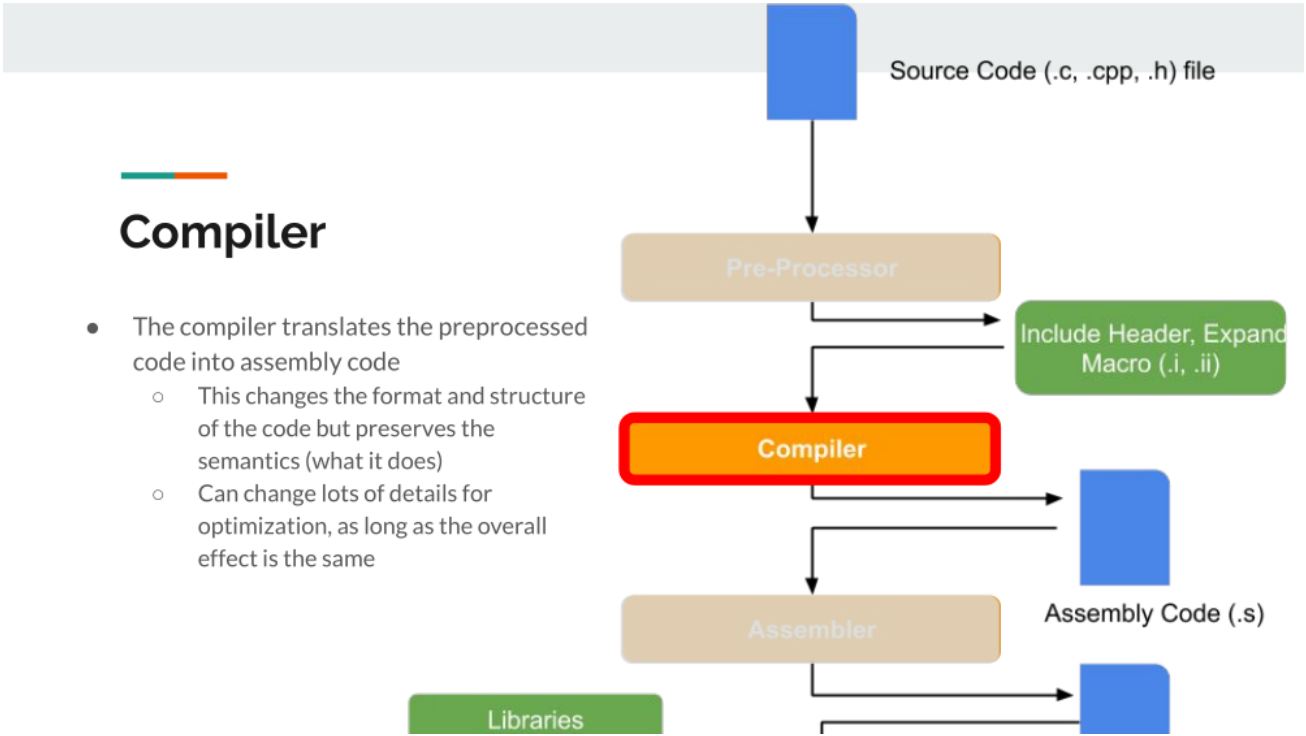


Figure 6

Before and after compilation

```c
extern int printf (const char *__restrict
                   __format, ...);
int main(void) {
    printf("CHAR_MIN = %d\n"
           "CHAR_MAX = %d\n",
           (-0x7f - 1), 0x7f);
    return 0;
}
```

- C *source code* converted to *assembly language*
- Textual, but 1:1 correspondence to machine language
- String out-of-line, referred to by label (`.LC0`)
- `printf` just referred to, not declared

```
    .file   "test.c"
    .section    .rodata.str1.1,"aMS",@progbits,1
.LC0:
    .string "CHAR_MIN = %d\nCHAR_MAX = %d\n"
    .text
    .globl  main
main:
    subq    $8, %rsp
    movl    $127, %edx
    movl    $-128, %esi
    leaq    .LC0(%rip), %rdi
    xorl    %eax, %eax
    call    printf@PLT
    xorl    %eax, %eax
    addq    $8, %rsp
    ret
    .size   main, .-main
```

Figure 7

C to Machine Code

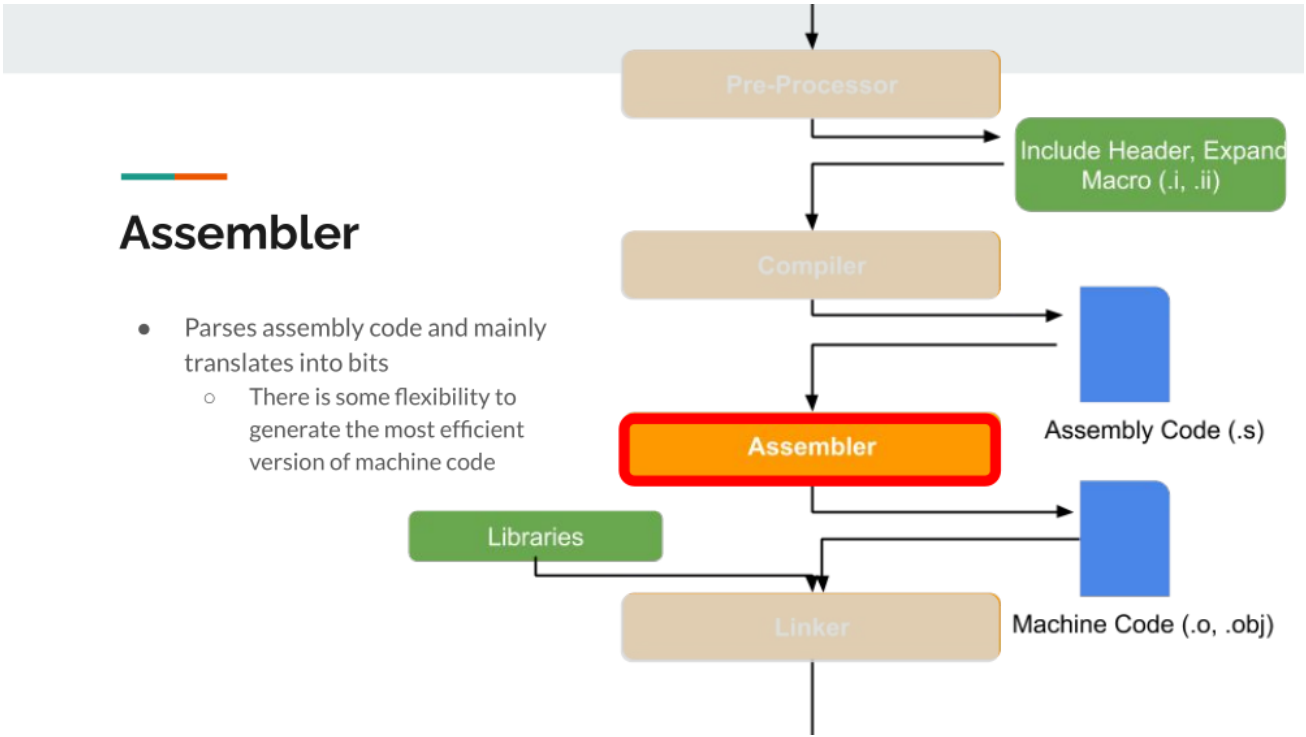


Figure 8

Before and after assembling

```
    .file   "test.c"
    .section    .rodata.str1.1,"aMS",@progbits,1
.LC0:
    .string "CHAR_MIN = %d\nCHAR_MAX = %d\n"
    .text
    .globl  main
main:
    subq    $8, %rsp
    movl    $127, %edx
    movl    $-128, %esi
    leaq    .LC0(%rip), %rdi
    xorl    %eax, %eax
    call    printf@PLT
    xorl    %eax, %eax
    addq    $8, %rsp
    ret
    .size   main, .-main
```

- Everything is now binary
- "Relocations" for addresses not yet known

```
$ objdump -s -r test.o
test.o: file format elf64-x86-64

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000011 R_X86_64_PC32     .LC0-0x0000000000000004
0000000000000018 R_X86_64_PLT32    printf-0x0000000000000004

Contents of section .rodata.str1.1:
 0000 43484152 5f4d494e 203d2025 640a4348  CHAR_MIN = %d.CH
 0010 41525f4d 4158203d 2025640a 00        AR_MAX = %d..

Contents of section .text:
 0000 4883ec08 ba7f0000 00be80ff ffff488d  H.............H.
 0010 3d000000 0031c0e8 00000000 31c04883  =....1......1.H.
 0020 c408c3                               ...
```

Figure 9

Before and after assembling

```
    .file   "test.c"
    .section    .rodata.str1.1,"aMS",@progbits,1
.LC0:
    .string "CHAR_MIN = %d\nCHAR_MAX = %d\n"
    .text
    .globl  main
main:
    subq    $8, %rsp
    movl    $127, %edx
    movl    $-128, %esi
    leaq    .LC0(%rip), %rdi
    xorl    %eax, %eax
    call    printf@PLT
    xorl    %eax, %eax
    addq    $8, %rsp
    ret
    .size   main, .-main
```

- Just to emphasize that 1:1 correspondence between assembly and machine instructions

```
$ objdump -d -r test.o
test.o:  file format elf64-x86-64
Disassembly of section .text.startup:

0000000000000000 <main>:
   0:   48 83 ec 08             sub    $0x8,%rsp
   4:   ba 7f 00 00 00          mov    $0x7f,%edx
   9:   be 80 ff ff ff          mov    $0xffffff80,%esi
   e:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi
                11: R_X86_64_PC32 .LC0-0x4
  15:   31 c0                   xor    %eax,%eax
  17:   e8 00 00 00 00          call   1c <main+0x1c>
                18: R_X86_64_PLT32    printf-0x4
  1c:   31 c0                   xor    %eax,%eax
  1e:   48 83 c4 08             add    $0x8,%rsp
  22:   c3                      ret
```

Figure 10

C to Machine Code

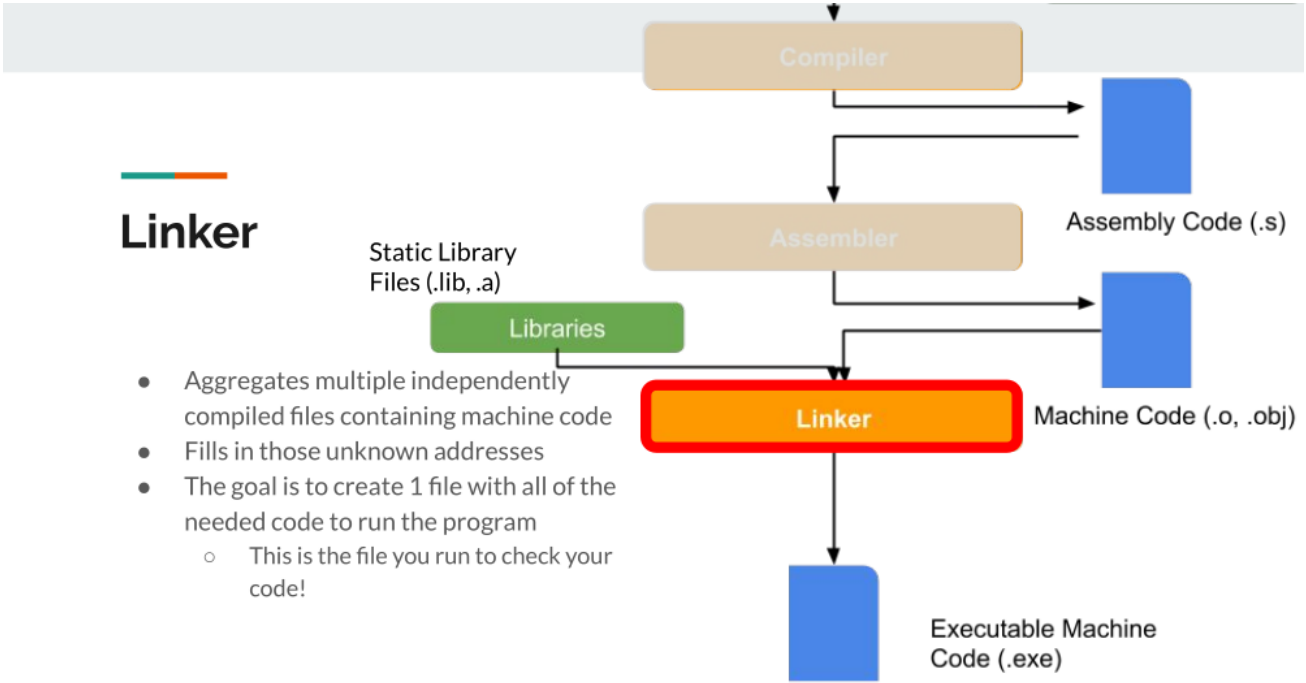


Figure 11

Cortex-M4 Memory Layout

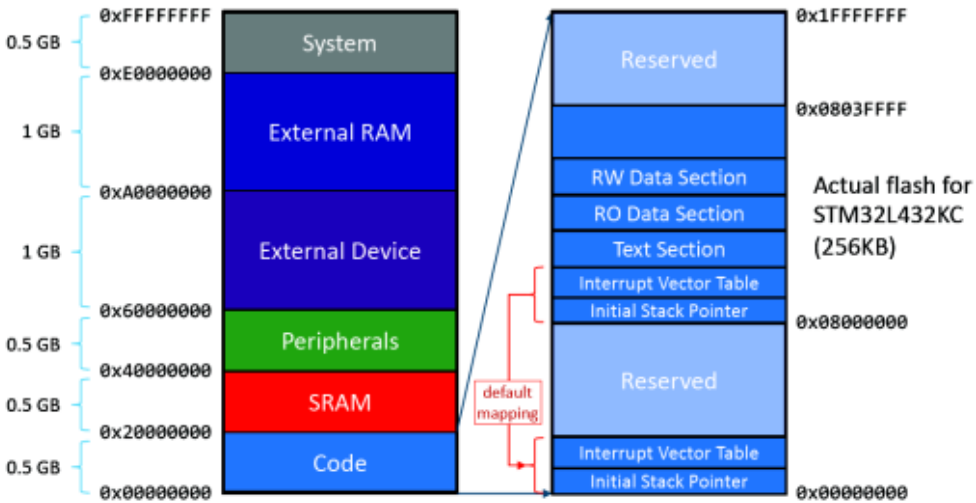


Figure 12

Useful website

For step by step installation:

- Setup Keil MDK:
  `https://www.youtube.com/watch?v=d_O2tu5CMbQ`
- Creating first project with keil uvision 5 ARM:
  `https://www.youtube.com/watch?v=JYMpyp3vtbY`