The Compilation Toolchain
Cross-Compilation for Embedded Systems

Prof. Andrea Marongiu
(andrea.marongiu@unimore.it)

Toolchain

The toolchain is a set of development tools used in association with source code or binaries generated from the source code

- Enables development in a programming language (e.g., C/C++)

- It is used for a lot of operations such as
  a) *Compilation*
  b) *Preparing Libraries*
  c) *Reading a binary file (or part of it)*
  d) *Debugging*

Most common toolchain is the GNU toolchain which is part of **the GNU project**

- Normally it contains
  a) *Compiler : Generate object files from source code files*
  b) *Linker: Link object files together to build a binary file*
  c) *Library Archiver: To group a set of object files into a library file*
  d) *Debugger: To debug the binary file while running*
  e) *And other tools*

The GNU Toolchain

GNU (GNU's Not Unix)

The GNU toolchain has played a vital role in the development of the Linux kernel, BSD, and software for embedded systems.

The GNU project produced a set of programming tools.

Parts of the toolchain we will use are:

-*gcc*: (GNU Compiler Collection): suite of compilers for many programming languages

-*binutils*: Suite of tools including linker (*ld*), assembler (*gas*)

-*gdb*: Code debugging tool

-libc: Subset of standard C library (assuming a C compiler).

-*bash*: free Unix shell (Bourne-again shell). Default shell on GNU/Linux systems and Mac OSX. Also ported to Microsoft Windows.

-*make*: automation tool for compilation and build

Program development tools

The process of converting source code to an executable binary image requires several steps, each with its own tool.

Compile:

C language source files (.c) are translated to **assembly files** (.s) by the *compiler* (*cc1)*

Assemble:

Assembly files are translated to **object files** (.o) by the *assembler* (*as*).

Link:

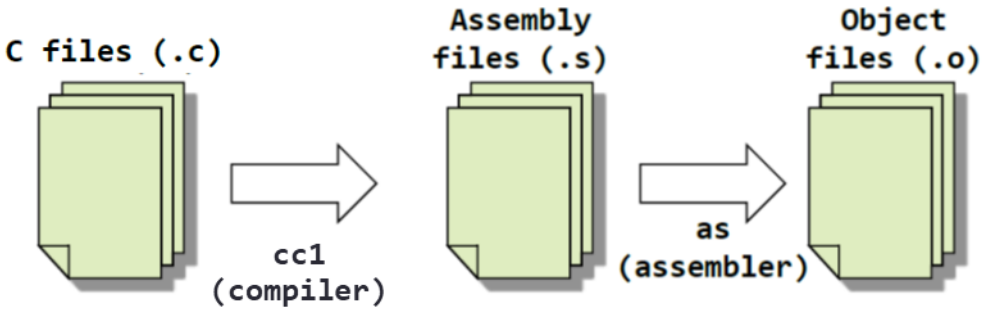
Object files produced by the compile/assemble steps are linked together by the *linker* (*ld*) to produce a single object file, called the **relocatable program**.

Relocate:

Physical memory addresses are assigned to the relative offsets within the relocatable program. This is also handled by *ld* (or the dynamic linker).

Program development tools

The output of the *assembler* is an *object* file. (*program.o*)



Object file:

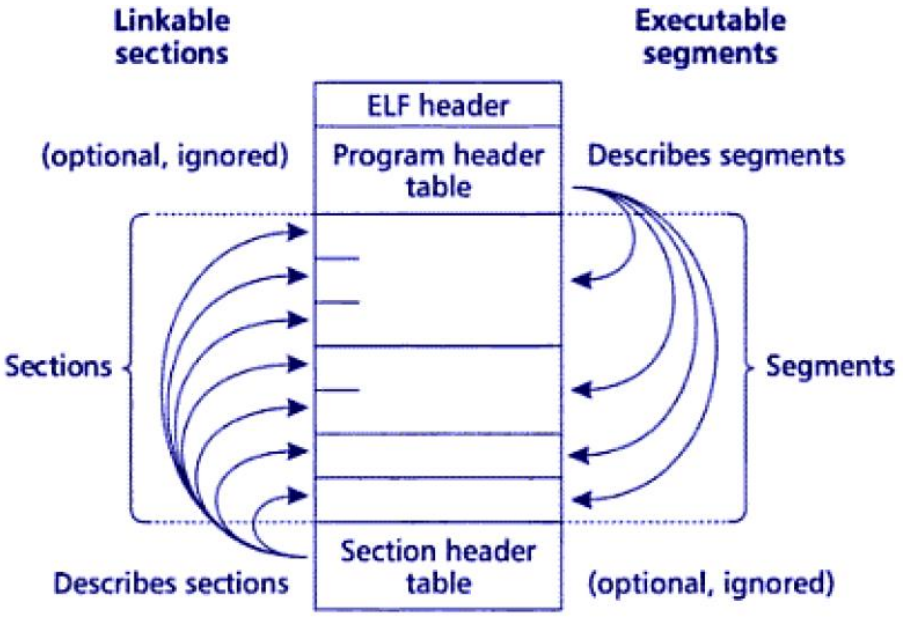
- a ***binary*** file that contains instructions and data from the language translation
- cannot be executed directly
- an incomplete image of the program

The *object file* contains separate ***sections*** containing code or data

- Specified according to the *Executable and Linkable Format (**ELF**)*

Executable and Linkable Format

- **The old one – a.out for UNIX community**
- **ELF**
  - standard for Linux
  - better support for cross-compilation, dynamic linking, initializer/finalizer
- **An ELF file can be one of the 3 types**
  - Relocatable
  - Executable
  - Shared object
- **Two views**
  - Compilers, assemblers, and linkers – a set of logical ***sections***.
  - System loader – a set of ***segments***



An ELF file has two views: the program header shows the *segments* used at run time, whereas the section header lists the set of *sections* of the binary

Executable and Linkable Format

Partial list of the ELF sections

- .init Startup
- .text The program
- .fini Shutdown
- .rodata Read Only Data
- .data Initialized Data
- .tdata Initialized Thread Data
- .tbss Uninitialized Thread Data
- .ctors Constructors
- .dtors Destructors
- .got Global Offset Table
- .bss Uninitialized Data

Executable and Linkable Format

- Besides the binary code of the program, ELF files contain other info such as a ***symbol table*** which contains a list of symbols generated during the compilation process along with their addresses in the file

- Symbols in the ***symbol table*** include (among other things):

  - Names of **static and global variables** defined in the file
  - Names of **static and global variables** used in the source file
  - **Functions** defined by the source file
  - **Functions** used (called) within the source file
  - Symbols used in the file can be either:

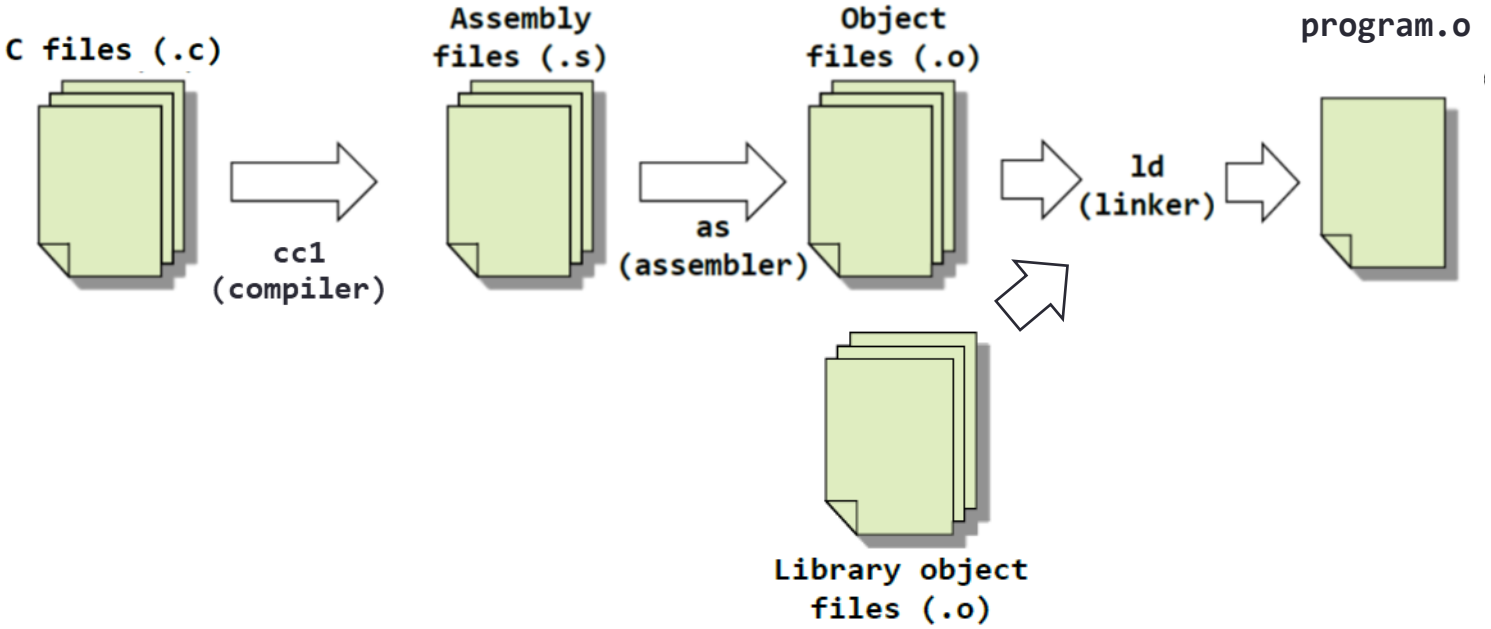
    - Defined in the same file (**defined symbols**)
    - Not defined in the file (**undefined symbols**)

Program development tools

- Often we have multiple “.c” files that result in multiple “.o” files.
- Each “.c” file was compiled into a separate “.o” file.
- Code in one file may call a function in code that is within another file. How can it know about the function? The compiler has to mark it to be resolved later.
  - The same applies to code implemented in third-party software or ***libraries***
- All the object files must be combined to resolve the symbol names. The ***linker*** (*ld*) accomplishes this.

Program development tools

- The output of the linker is a new object file (*program.o*) that contains all the code and data from all the (subprogram.o) files. It merges all the *.text*, *.data*, *.bss* sections.



Program development tools

The process of converting source code to an executable binary image requires several steps, each with its own tool.

Compile:

C language source files (.c) are translated to **assembly files** (.s) by the *compiler* (*cc1)*

Assemble:

Assembly files are translated to **object files** (.o) by the *assembler* (*as*).

Link:

Object files produced by the compile/assemble steps are linked together by the *linker* (*ld*) to produce a single object file, called the **relocatable program**.

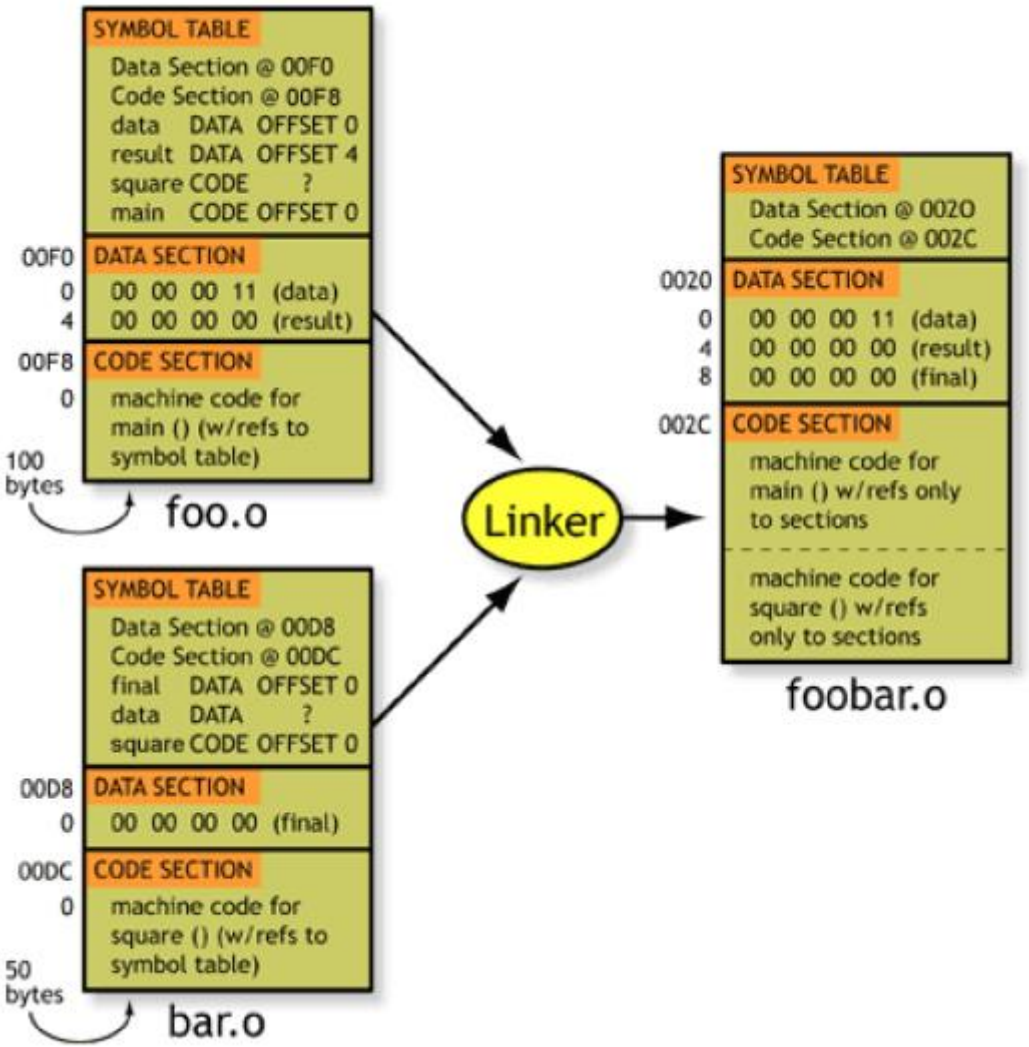
Relocate:

Physical memory addresses are assigned to the relative offsets within the relocatable program. This is also handled by *ld*.

The linking process

- The linker job is to connect the dots by combining multiple object files using the symbol table in each one of them
  - If the used symbol is defined in the same file, then replaces the symbol with an address of its location
  - If the used symbol is undefined in the same file, then it looks for the address of that symbol in other object files
    - Object files generated from other source files
    - Library files passed to the Linker
- The outcome of the linking process is an executable that does not rely on symbols, since all of them are replaced by addresses (resolved)
- Hence, the symbol table is essential for the linking process, and is not needed to run the executable
  - However, it can be useful when running the debugger

The linking process



Dealing with libraries

- Normally the program functionality are located in:
  - The program source code (functionality specific to this program)
  - Some pre-compiled Libraries (functionality that is used by multiple programs, such as printing, writing to files, …)

- Hence the program needs to be linked to some libraries in addition to the object files of the source files of the program

- Libraries can be,
  - **Static Libraries**:
    - A static library is a simple archive of pre-compiled object files
    - Linking of a static library occurs at the same time of object files of the program
    - The library becomes a part of the executable image

  - **Dynamic Libraries** (shared objects):
    - Linking occurs at run time
    - Hence the executable image does not contain the required functionality
    - The shared object should be available to the executable at run time

Static VS Dynamic Linking

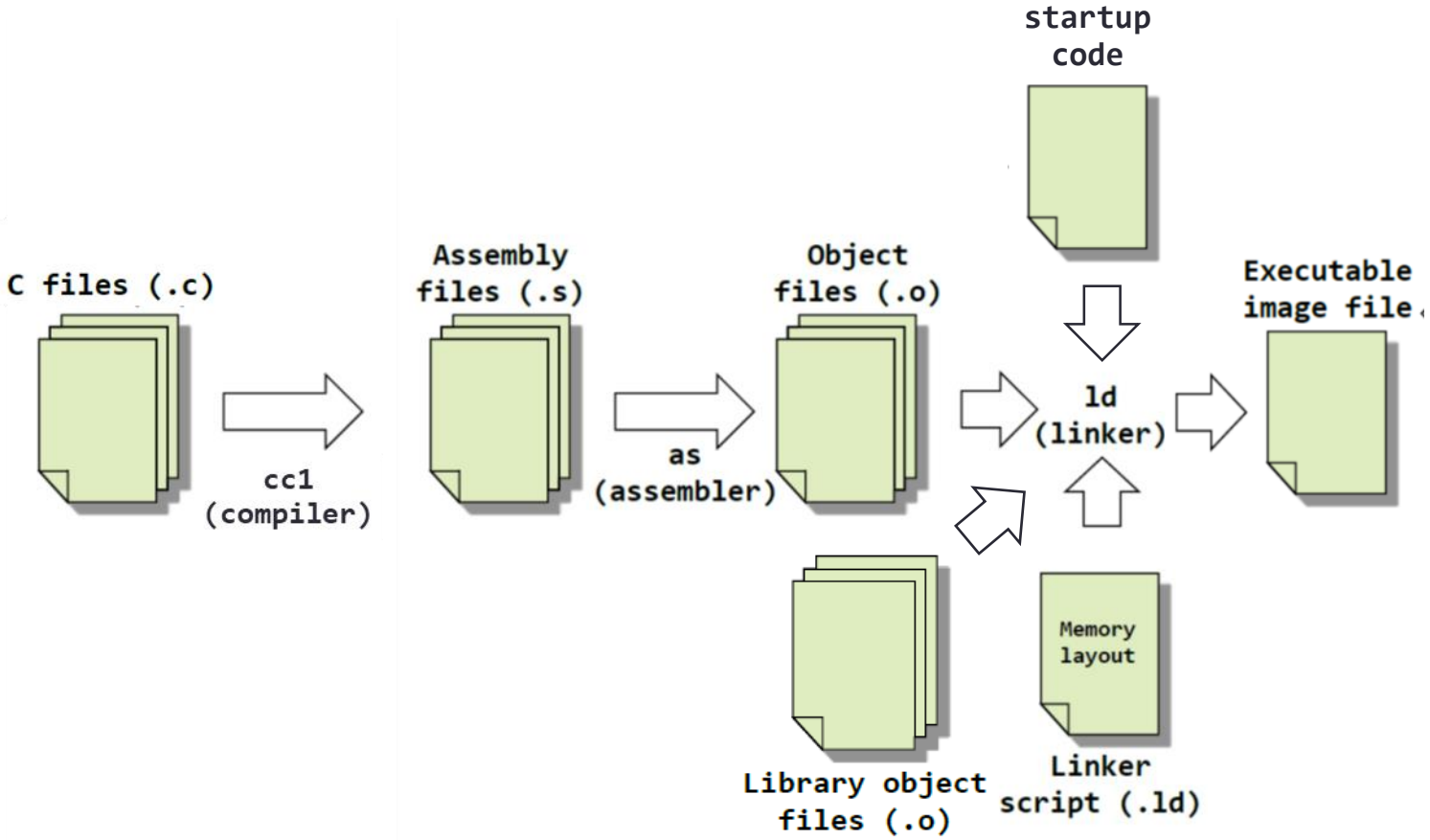
- **Why do we use Dynamic Linking?**
  - To keep the executable binary image size smaller
  - Since multiple programs may be using the same library, it makes no sense to include it in each and every one of them
  - Also, as the different programs loaded in memory, they need less memory (since the shared object will be loaded only once in the memory)
  - Upgrade in functionality and bug fixing of the library does not require re-building of all the programs using this library

- **Why do we use Static Linking?**
  - To remove dependencies (the program has everything it needs to run)
  - To make sure we are using a specific version of the library (to avoid conflicts)
  - Used normally with libraries that are not common to be used by other programs

Program development tools

To produce the final executable the *linker* needs two more bits of information:

- ***Linker script***: A text file that describe how the sections in the input files should be mapped into the output file, and to control the memory layout of the output file.
- ***Startup code***: a small block of assembly code that prepares the way for the execution of software written in a high-level language.



Linker script

- **Describes the memory layout and how to map sections in the memory map.**
  - **ENTRY** linker script command to set the entry point.
  - **SECTIONS** command to describe memory layout
  - **`.'** indicates the location counter
  - **MEMORY** command describes the location and size of blocks of memory in the target

```
SECTIONS
{
        .text.start (_KERNEL_BASE_) : {
              startup.o( .text )
        }

        .text : ALIGN(0x1000) {
              _TEXT_START_ = .;
              *(.text)
              _TEXT_END_ = .;
        }

        .data : ALIGN(0x1000) {
              _DATA_START_ = .;
              *(.data)
              _DATA_END_ = .;
         }

        .bss : ALIGN(0x1000) {
              _BSS_START_ = .;
              *(.bss)
              _BSS_END_ = .;
         }
}
```

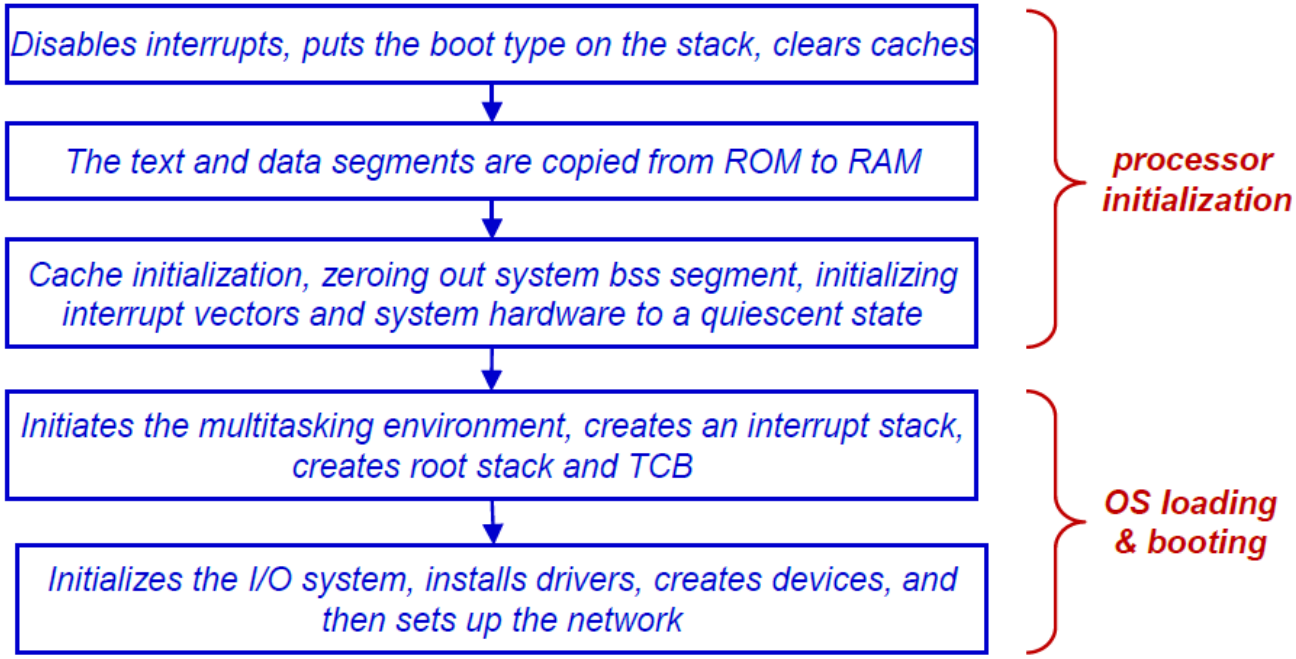
```
MEMORY
{
        ROM (rx) : ORIGIN = 0, LENGTH = 256k
        RAM (wx) : org = 0x00100000, len = 1M
}
```

Startup code

- Another job the linker does is to insert *startup* code, a small block of assembly code that prepares the way for the execution of software written in a high-level language (possibly initiating a *booting sequence*).

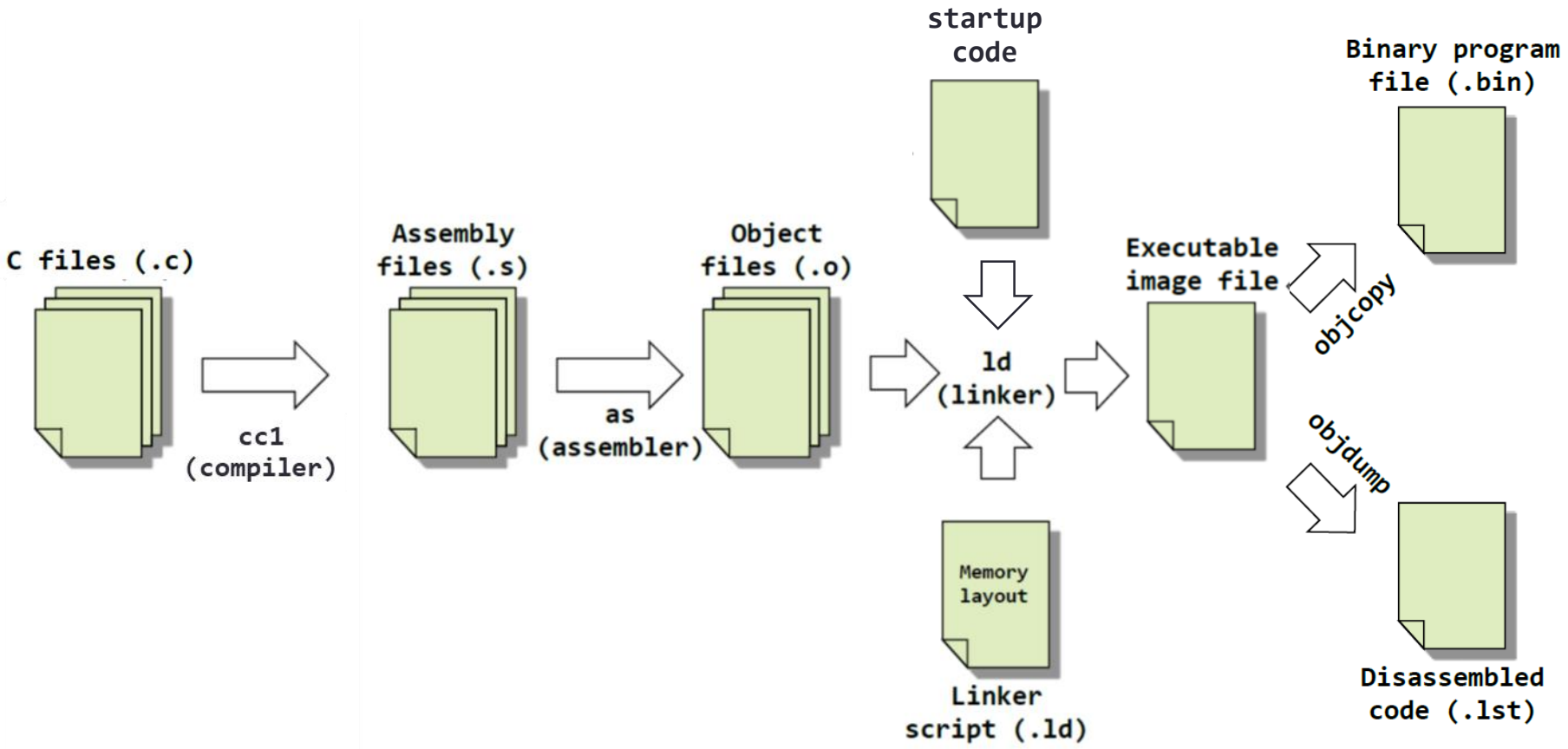
- For example, C programs expect a *stack* to exist. Space for the *stack* must be allocated before the C code can be run.

- *Startup code* for C programs usually consist of the following series of actions:
  - disable interrupts
  - copy initialized data from ROM to RAM
  - zero out the uninitialized data area
  - allocate space for and initialize the *stack*
  - call **`main()`**;



Program development tools

- Besides the tools to build a program, a toolchain contains tools to copy/convert the contents of an object file from one format to another (***objcopy***) and to disassemble an executable file (***objdump***).



Toolchain contents (summary)

gcc
(build tools driver)

`cc1`	*C compiler*	**COMPILER**
`as`	*assembler*	**BINUTILS**
`ld`	*linker*	
`objcopy`	*format converter*	
`objdump`	*disassembler*	
`ar`	*archiver*	
`readelf`	*ELF reader*	
`size`		
`strip`		
`nm`		
`nlmconv`		
`ranlib`		
`gdb`	*debugger*	**DEBUGGER**

Binutils

- **as** is the assembler and it converts human-readable assembly language programs into binary machine language code. It typically takes as input .s assembly les and outputs .o object files.
- **ld** is the linker and it is used to combine multiple object files by resolving their external symbol references and relocating their data sections, and outputting a single executable file. It typically takes as input .o object files and .ld linker scripts and outputs .out executable files.
- **objcopy** is a translation utility that copies and converts the contents of an object file from one format (e.g., .out) another (e.g., .hex, .bin).
- **objdump** is a disassembler but it can also display various other information about object files. It is often used to disassemble binary files (e.g. .out) into a canonical assembly language listing (e.g. .lst).

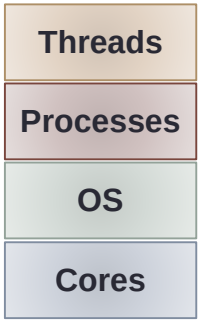
Binutils (2)

- **ar** is a utility for creating, modifying and extracting from archives.
- **nlmconv** converts object code into an NLM.
- **nm** lists symbols from object files.
- **ranlib** generates an index to the contents of an archive.
- **readelf** displays information from ELF-format object file.
- **size** displays the sections of an object or archive, and their sizes.
- **strip** Discards symbols embedded in object files.

Cross-compilation

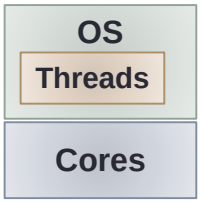
- Many parts of the build process (linker script, startup code, libraries) are specific to the target platform for which a program is being built
  - Depends of the availability of an OS, its type, and the thread model

- Toolchain: just the compiler, binutils and C library
- SDK: a toolchain, plus a number (potentially large) of libraries built for the target architecture, and additional native tools helpful when building software.



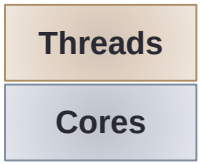
Full-fledged OS
(e.g. Windows, Linux)

General purpose computing



Lightweight OS
with embedded threads
(e.g., FreeRTOS, ARM MBed)

Real-time computing



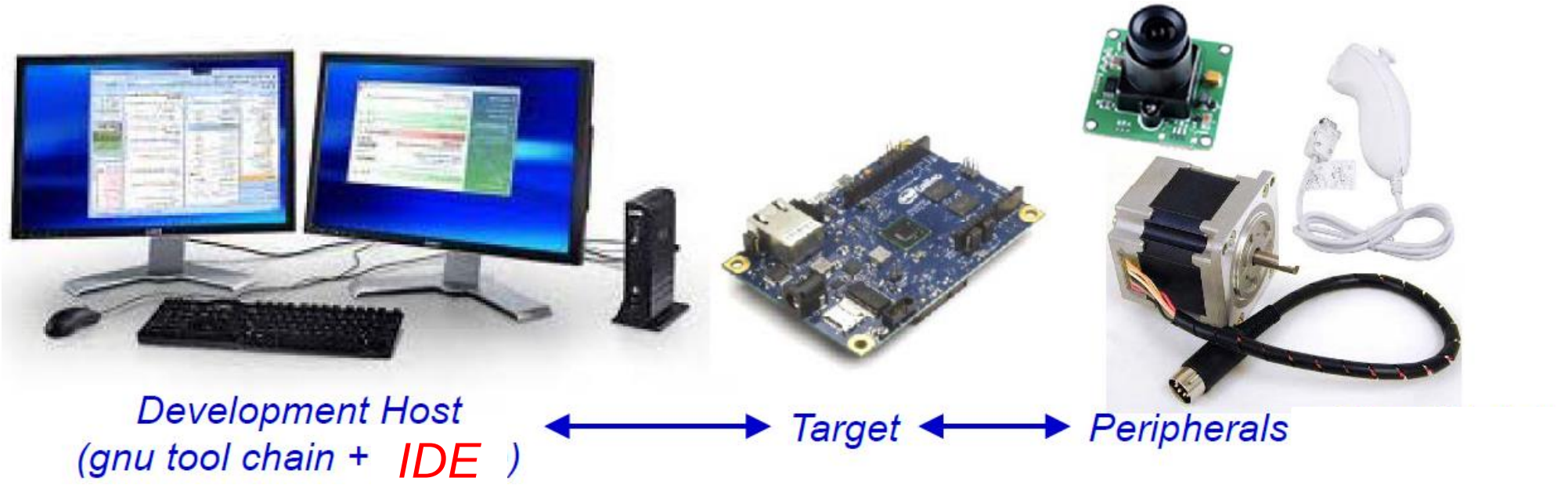
Bare metal
with threads
from HAL/SDK

Embedded computing

- When developing applications for embedded systems the build process is typically carried out onto a different, more powerful computer (*host*)

Cross-compilation

- Many parts of the build process (linker script, startup code, libraries) are specific to the target platform for which a program is being built
  - Depends of the availability of an OS, its type, and the thread model



- Use the ***host*** to edit, compile, and build application programs
- At the ***target***, use tools to load, execute, debug, and monitor

Cross-compilation toolchain

- A set of tools that allows to build source code into binary code for a target platform different than the one where the build takes place
  - Different CPU architecture
  - Different **ABI**
  - Different operating system
  - Different C library
- Three machines involved in the build process
  - **build** machine, where the build takes place
  - **host** machine, where the execution takes place
  - **target** machine, for which the programs generate code
- Native toolchain: build == host == target
- Cross-compilation toolchain: build == host != target
- Corresponds to the --build, --host and --target *autoconf* configure script arguments

Cross-compilation toolchain

○ ABI definition

- ABI = *Application Binary Interface*
- From the point of a toolchain, the ABI defines:
  - How function calls are made (so-called *calling convention*)
    - How arguments are passed: in registers (which ones?), on the stack, how 64-bits arguments are handled on 32 bits architectures
    - How the return value is passed
  - Size of basic data types
  - Alignment of members in structures
  - When there is an operating system, how system calls are made
- Object files from different ABIs cannot be linked together (especially important if you have pre-built libraries or executables!)
- For a given CPU architecture, there can potentially be an infinite number of ABIs: ABIs are just specifications on how to use the CPU architecture
- Need to understand the ABIs for each architecture.

Cross-compilation toolchain

- *autoconf* defines the concept of system definitions, represented as tuples
- A system definition describes a system: CPU architecture, operating system, vendor, ABI, C library
- Different forms:
  - <arch>-<vendor>-<os>-<libc/abi>, full form
  - <arch>-<os>-<libc/abi>
- Components:
  - <arch>, the CPU architecture: arm, mips, powerpc, i386, i686, etc.
  - <vendor>, (mostly) free-form string, ignored by autoconf
  - <os>, the operating system. Either none or linux for the purpose of this talk.
  - <libc/abi>, combination of details on the C library and the ABI in use

Cross-compilation toolchain

Toolchain tuple examples

`arm-foo-none-eabi`	bare-metal toolchain targeting the ARM architecture, from vendor foo
`arm-unknown-linux-gnueabihf`	Linux toolchain targeting the ARM architecture, using the EABIhf ABI and the glibc C library, from an unknown vendor
`armeb-linux-uclibcgnueabi`	Linux toolchain targeting the ARM big-endian architecture, using the EABI ABI and the uClibc C library
`mips-img-linux-gnu`	Linux toolchain targeting the MIPS architecture, using the glibc C library, provided by Imagination Technologies

Cross-compilation toolchain

- **Bare-metal VS Linux toolchain**

- *Two main values for <os>*
  - ***none** for bare-metal toolchains*
    - Used for development without an operating system
    - C library used is generally newlib
    - Provides C library services that do not require an operating system
    - Allows to provide basic system calls for specific hardware targets
    - Can be used to build bootloaders or the Linux kernel, cannot build Linux userspace code
  - ***linux** for Linux toolchains*
    - Used for development with a Linux operating system
    - Choice of Linux-specific C libraries: glibc, uclibc, musl
    - Supports Linux system calls
    - Can be used to build Linux userspace code, but also bare-metal code such as bootloaders or the kernel itself

Cross-compilation toolchain

- *There are four core components in a Linux cross-compilation toolchain*

1. **binutils**
   - *Needs to be configured for each CPU architecture: your native x86 binutils cannot produce ARM code.*

2. **gcc**
   - *Provides:*
     a) The compiler itself, cc1 for C, cc1plus for C++. Only generates assembly code in text format.
     b) The compiler driver, gcc, g++, which drives the compiler itself, but also the binutils assembler and linker.
     c) Target libraries: libgcc (gcc runtime), libstdc++ (the C++ library), libgfortran (the Fortran runtime)
     d) Header files for the standard C++ library.

   - *Building gcc is a bit more involved than building binutils*

3. **Linux kernel headers**
   - *Needed to build a C library **if a Linux OS is available on the target**: contain definitions of system call numbers, various structure types and definitions.*

4. **C library**

Cross-compilation toolchain

○ C library

- Provides the implementation of the POSIX standard functions, plus several other standards and extensions
- Based on the Linux system calls
- Several implementations available:
  - glibc
  - uClibc-ng (formerly uClibc)
  - musl
  - bionic, for Android systems
  - A few other more special-purpose: newlib (for bare-metal), dietlibc, klibc
- After compilation and installation, provides:
  - The dynamic linker, `ld.so`
  - The C library itself `libc.so`, and its companion libraries: `libm`, `librt`, `libpthread`, `libutil`, `libnsl`, `libresolv`, `libcrypt`
  - The C library headers: `stdio.h`, `string.h`, etc.

Cross-compilation toolchain

- **Concept of SYSROOT**

  - The *sysroot* is the **the logical root directory for headers and libraries**
  - Where *gcc* looks for headers, and *ld* looks for libraries
  - Both *gcc* and *binutils* are built with `--with-sysroot=<SYSROOT>`
  - The kernel headers and the C library are installed in `<SYSROOT>`
  - If the toolchain has been moved to a different location, *gcc* will still find its sysroot if it's in a subdir of `--prefix`
    - `--prefix=/home/thomas/buildroot/arm-uclibc/host/usr`
    - `--with-sysroot=/home/thomas/buildroot/arm-uclibc/host/usr/arm-buildroot-linux-uclibcgnueabihf/sysroot`
  - Can be overridden at runtime using gcc's `--sysroot` option.
  - The current *sysroot* can be printed using the `-print-sysroot` option.

Cross-compilation toolchain

- **Toolchain SYSROOT organization**

  - arm-buildroot-linux-uclibcgnueabihf/
  - bin/
  - include/
  - lib/
  - libexec/
  - share/

/home/hero-vm/hero-sdk/hero-gcc-toolchain/install

Cross-compilation toolchain

- **Toolchain SYSROOT organization**

- `arm-buildroot-linux-uclibcgnueabihf/`
  - `bin/`
    - Limited set of *binutils* programs, without their cross-compilation prefix. Hard links to their counterparts with the prefix. This is where *gcc* finds them.
  - `include/c++/4.9.4/`
    - Headers for the C++ standard library, installed by *gcc*
    - Interestingly, they are not part of the *sysroot* per-se.
  - `lib/`
    - The *gcc* runtime libraries, built for the target
    - `libatomic`, provides a software implementation of `atomic` built-ins, when needed
    - `libgcc`, the main *gcc* runtime (optimized functions, 64-bit division, floating point emulation)
    - `libitm`, transactional memory library
    - `libstdc++`, standard C++ library
    - `libsupc++`, subset of `libstdc++` with only the *language support* functions

Cross-compilation toolchain

- **Toolchain SYSROOT organization**

- `bin/`
  - `arm-buildroot-linux-uclibcgnueabihf-` prefixed tools
  - From *binutils*: addr2line, ar, as, elfedit, gcov, gprof, ld, nm, objcopy, objdump, ranlib, readelf, size, strings, strip
  - From *gcc*: c++ (same as g++), cc (same as gcc), cpp, g++, gcc, gcc-ar, gcc-nm, gcc-ranlib
  - The `gcc-{ar,nm,ranlib}` are wrappers for the corresponding binutils program, to support Link Time Optimization (LTO)

- `include/`
  - Headers of the host libraries (*gmp*, *mpfr*, *mpc*)

- `share/`
  - documentation (man pages and info pages)
  - translation files for gcc and binutils

Cross-compilation toolchain

Toolchain SYSROOT organization

- `lib/`
  - `gcc/arm-buildroot-linux-uclibcgnueabihf/4.9.4/`
    - `crtbegin*.o`, `crtend*.o`, object files handling constructors/destructors, linked into executables
    - `include/`, headers provided by the compiler (`stdarg.h`, `stdint.h`, `stdatomic.h`, etc.)
    - `include-fixed/`, system headers that gcc fixed up using *fixincludes*
    - `install-tools/`, also related to the *fixincludes* process
    - `libgcc.a`, `libgcc_eh.a`, `libgcov.a`, static variants of the gcc runtime libraries
  - `ldscripts/`, linker scripts provided by *gcc* to link programs and libraries
  - Host version of *gmp*, *mpfr*, *mpc*, needed for *gcc*
- `libexec/`
  - `gcc/arm-buildroot-linux-uclibcgnueabihf/4.9.4/`
    - `cc1`, the actual C compiler
    - `cc1plus`, the actual C++ compiler
    - `collect2`, program from gcc collecting initialization functions, wrapping the linker
    - `install-tools/`, misc *gcc* related tools, not needed for the compilation process
    - `liblto_plugin.so.0.0.0`, `lto-wrapper`, `lto1`, related to LTO support (outside of the scope of this talk)

Cross-compilation toolchain

- **C library: size comparison**

	glibc	uclibc	musl
ld, dynamic linker	121 KB	25 KB	N/A
libc	878 KB	286 KB	437 KB
libcrypt	30 KB	17 KB	N/A
libdl	9.5 KB	9 KB	N/A
libm	414 KB	37 KB	N/A
libnsl	54 KB	4.7 KB	N/A
libnss_dns	14 KB	N/A	N/A
libnss_files	30 KB	N/A	N/A
libpthread	105 KB	76 KB	N/A
libresolv	54 KB	4.7 KB	N/A
librt	22 KB	13 KB	N/A
libutil	9.5K	4.7 KB	N/A
TOTAL	1741 KB	477 KB	437 KB

ARM Cortex-A9 toolchain built with the Thumb-2 instruction set, using Buildroot. gcc 4.9, binutils 2.26, musl 1.1.15, glibc 2.23, uclibc-ng 1.0.17

Exercise – code snippet 1

- Let's start from the simplest piece of code

include the declarations needed to invoke the *printf* C library (*libc*) function.

```
#include <stdio.h>

int main(void)
{
    printf("Hello World!\n");
    return 0;
}
```

declares function *main*, which is believed to be our program entry point (is it?).

no parameters, returns an integer

the parent process --- the *shell* – expects a child process returns an 8-bit number representing its status:

- *0* for normal termination
- *0 < n < 128* for abnormal termination
- *n > 128* for signal-induced termination.

definition of function *main*, which invokes the *printf* C library function to output the "Hello World!\n" string and returns 0 to the parent process.

Exercise – code snippet 2

- Second piece of code, variable allocation.

```
int a;
int b = 10;
int * c;
int d[10];
int e[10] = {0,1,2,3,4,5,6,7,8,9};
//const int f;
const int g = 10;

int main(void){
    int h;
    int i = 10;
    int * j;
    int k[10];
    int l[10] = {0,1,2,3,4,5,6,7,8,9};
    //const int m;
    const int n = 10;
    int * o;
    int * p = new int;
    int * q = new int(10);
    int * r = new int[10];
    static int s;
    static int t = 10;
}
```

Global initialized and uninitialized variables and constants

Local initialized and uninitialized variables and constants (statically and dynamically)

Static variables

Exercise 1 – The `gcc` toolchain driver

The `gcc` command is in fact a driver, that invokes one after the other all the tools that are required to produce an executable out of a source file.
Try various flags:

- -E – to stop the driver after the preprocessing stage (the output is a C file)
- -S – to stop the driver after the compilation stage (the output is an asm file)
- -c – to stop the driver after the assembling stage (the output is an obj file)
- -o – to produce the final executable (default name is a.out)
- -v – to show (verbose mode) how the driver does all the magic for us

- **Apply to code snippets 1 and 2**

Exercise 1 – The `binutils` tools

Run the **`file`** tool on the compiled variants from the previous exercise

- How does the message differ between –c and global compilation?

```
hero-vm@ubuntu: ~/hero-sdk/hero-openmp-examples/variables
hero-vm@ubuntu:~/hero-sdk/hero-openmp-examples/variables$ file var.o
var.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
hero-vm@ubuntu:~/hero-sdk/hero-openmp-examples/variables$ file a.out
a.out: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/l, for GNU/Linux 2.6.32, BuildID[sha1]=76c60bc0d8d238283634d3
08ea2f62cafeebe7e3, not stripped
```

Try the **`file`** tool for different toolchains

- Try it on native host (x86) toolchain
- Try it on ARM cross-toolchain
- Try it on RISCV cross-toolchain

```
hero-vm@ubuntu:~/hero-sdk/hero-openmp-examples/variables$ riscv32-unknown-elf-g+
+ -c var.c -o varR.o
hero-vm@ubuntu:~/hero-sdk/hero-openmp-examples/variables$ arm-linux-gnueabihf-g+
+ var.c -c -o varA.o
hero-vm@ubuntu:~/hero-sdk/hero-openmp-examples/variables$ file var
varA.o  var.c   var.o   varR.o
hero-vm@ubuntu:~/hero-sdk/hero-openmp-examples/variables$ file var.o
var.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
hero-vm@ubuntu:~/hero-sdk/hero-openmp-examples/variables$ file varA.o
varA.o: ELF 32-bit LSB relocatable, ARM, EABI5 version 1 (SYSV), not stripped
hero-vm@ubuntu:~/hero-sdk/hero-openmp-examples/variables$ file varR.o
varR.o: ELF 32-bit LSB relocatable, UCB RISC-V, version 1 (SYSV), not stripped
```

Exercise 1 – The `binutils` tools

Try the following tools

- **nm** – lists symbols from obj files

```
hero-vm@ubuntu:~/hero-sdk/hero-openmp-examples/variables$ nm var.o
0000000000000000 B a
0000000000000000 D b
0000000000000008 B c
0000000000000020 B d
0000000000000020 D e
0000000000000000 T main
                 U __stack_chk_fail
0000000000000000 r _ZL1g
                 U _Znam
                 U _Znwm
0000000000000048 b _ZZ4mainE1s
0000000000000048 d _ZZ4mainE1t
```

USEFUL FLAGS:

- -u – show undefined symbols only
- --defined-only (-D)

SYMBOL TYPES

A : Global absolute symbol.
a : Local absolute symbol.
B : Global bss symbol.
b : Local bss symbol.
D : Global data symbol.
d : Local data symbol.
f : Source file name symbol.
L : Global thread-local symbol (TLS).
l : Static thread-local symbol (TLS).
T : Global text symbol.
t : Local text symbol.
U : Undefined symbol.

Exercise 1 – The `binutils` tools

- **`objdump`** – displays various information from obj files
  - -h – contents of the section headers
  - -p – contents of the program header + dynamic section
  - -r/R – static/dynamic relocation records
  - -t – symbol table
  - -d/D – disassemble (executable sections/all)
  - -S – intermix source code with disassembly

- **`objdump -hrt`** to show sections, symbols and static relocation
- **`objdump -D`** to show disassembled code

Exercise 2 – The complete process

Let's compile **code snippet 1**

- **`gcc -Os -c hello.c`**
- **`file hello.o`**
- **`objdump -hrt`** hello.o

```
hero-vm@ubuntu:~/hero-sdk/hero-openmp-examples/hello$ objdump -hrt hello.o

hello.o:     file format elf64-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         00000015  0000000000000000  0000000000000000  00000040  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000000  0000000000000000  0000000000000000  00000055  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  0000000000000000  0000000000000000  00000055  2**0
                  ALLOC
  3 .rodata       0000000d  0000000000000000  0000000000000000  00000055  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .comment      00000036  0000000000000000  0000000000000000  00000062  2**0
                  CONTENTS, READONLY
  5 .note.GNU-stack 00000000  0000000000000000  0000000000000000  00000098  2**0
                  CONTENTS, READONLY
  6 .eh_frame     00000038  0000000000000000  0000000000000000  00000098  2**3
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 hello.c
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l    d  .rodata        0000000000000000 .rodata
0000000000000000 l    d  .note.GNU-stack        0000000000000000 .note.GNU-stack
0000000000000000 l    d  .eh_frame      0000000000000000 .eh_frame
0000000000000000 l    d  .comment       0000000000000000 .comment
0000000000000000 g     F .text  000000000000015 main
0000000000000000         *UND*  0000000000000000 puts


RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000005 R_X86_64_32       .rodata
000000000000000a R_X86_64_PC32     puts-0x0000000000000004


RELOCATION RECORDS FOR [.eh_frame]:
OFFSET           TYPE              VALUE
0000000000000020 R_X86_64_PC32     .text
```

1. **.text**: that's "Hello World" compiled program, i.e. IA-32 opcodes corresponding to the program. This will be used by the program *loader* to initialize the *process' code segment*.
2. **.data**: no initialized global variables nor initialized static local variables, so this section is empty.
3. **.bss**: no non-initialized variable, either global or local, so this section is also empty. (data allocated here is zeroed).
4. **.rodata**: contains the "Hello World!\n" string, tagged read-only.
5. **.comment**: this segment contains 33 bytes of comments which cannot be tracked back to our program, since we didn't write any comment. We'll soon see where it comes from.

Exercise 2 – The complete process

- Let's take a look at the asm code
- **`objdump -D hello.c`**
- **`gcc -S hello.c`**

```
        .file   "hello.c"
        .section        .rodata
.LC0:
        .string "Hello World!"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $.LC0, %edi
        call    puts
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.11) 5.4.0 20160609"
        .section        .note.GNU-stack,"",@progbits
~
```

1. From the assembly code, it becomes clear where the ELF section flags come from.
2. It also reveals where the .comment section comes from (second last line). Since printf was called to print a single string, and we requested our nice compiler to optimize the generated code (-Os), puts was generated instead.
3. And what about the assembly code produced? No surprises here: a simple call to function puts with the string addressed by .LC0 as argument.

Exercise 2 – The complete process

- Let's try linking to produce the final executable
- **`ld hello.o -lc`**

C Run Time 0 - CRT0

- `CRT0.o`
  - Initialize the executable file
    - stack pointer
    - .bss section
  - Defines special symbols like _start
  - Calls main()

- Something went wrong…
- Can we try manually linking those pieces?
- /usr/lib/x86_64-linux-gnu/crt1.o
  /usr/lib/x86_64-linux-gnu/crti.o
  /usr/lib/x86_64-linux-gnu/crtn.o

- Also, we need –lgcc
- remember –L to point to library paths

Linker Script

- Text file with some configurations
- Memory map for the application
  - Address of Sections
  - Informations to ELF Header
- Features of different (micro)Processors
- Attributes of each section - read/write

```
# ld --verbose
```

Exercise 2 – The complete process

- Looks like something is still missing…
- **`gcc -v hello.o`**

- Many more pieces in a sophisticated toolchain
- How about the size of the executable compared to our bits?

```
hero-vm@ubuntu:~/hero-sdk/hero-openmp-examples/hello$ ll
total 32
drwxrwxr-x  2 hero-vm hero-vm 4096 May 21 13:19 ./
drwxrwxr-x 12 hero-vm hero-vm 4096 May 21 12:44 ../
-rwxrwxr-x  1 hero-vm hero-vm 8600 May 21 13:10 a.out*
-rw-rw-r--  1 hero-vm hero-vm   83 May 21 12:45 hello.c
-rw-rw-r--  1 hero-vm hero-vm 1504 May 21 12:46 hello.o
-rw-rw-r--  1 hero-vm hero-vm  457 May 21 12:58 hello.s
```

- Play with nm and/or objdump -d to get an idea of what got linked into the executable.

Exercise 3 – Variable allocation

- Let's repeat all the steps with **code snippet 2**
- `g++ -O0 -c var`
- `objdump -hrt var.o`

.text: the compiled program, i.e. x86 opcodes corresponding to the program. This will be used by the program *loader* to initialize the *process' code segment*.
.data: initialized global variables and initialized static local variables. In this case, the *.data* section contain the variable initial values to be loaded into the *data segment*.
.bss: non-initialized variables, global and local. So, it indicates how many bytes must be allocated and zeroed in the *data segment* in addition to section *.data*.
.rodata: constant values, tagged read-only.
.comment: this segment contains 33 bytes of comments which cannot be tracked back to our program, since we didn't write any comment. We'll soon see where it comes from.

Exercise 3 – Variable allocation

- Let's look at the asm code
- **`g++ -S var.c`**

a: the variable "a" is in .bss section, because is global and is a non-initialized variable. Because variables that must be initialized with zeros are placed in .bss. Variable "a" is non-initialized but is global, and global variables non-initialized are initialized with zeros by default.
b: the variable "b" is in .data section, because is a global initialized variable
c: the variable pointer "c" is in .bss section, because the same reason of variable "a"
d: the array "d" is in .bss section, because the sabe reason of variable "a"
e: the array "e" is in .data section, beucase is an initialized array of integers
g: the constant "g" is in .rodata, which is destined to constants and is marked as read only.
h, **i**, **j**, **k**, **l**, **n** and **o**: These variables, initialized or not, will be placed on the program stack. The variables initialized will be initialized. Otherwise, if a variable is non-initialized, the SO will not do anything with the correspondent memory position content, so, maybe there is memory trash in this variables.
p, **q** and **r**: All these pointers will be placed in the program stack, however, the compiler will insert same instructions to allocate memory on the heap and initialize these variables if it necessary.
s: the variable "s" is in .bss. The reason is similar to non-initialized global variables.
t: the variable "t" is in .data. The reason is similar to initialized global variables.

More exercises

- Play with objdump –hrt
- Between native and cross toolchains
- Between file obtained with –c and no flags
- Understand
- - what happens to relocatable symbols
- - what happens to data symbols (where are placed)