Universität Bamberg

Operating Systems Engineering

Lecture 2: From Source to Boot

Michael Engel (michael.engel@uni-bamberg.de)
Lehrstuhl für Praktische Informatik, insb. Systemnahe Programmierung
https://www.uni-bamberg.de/sysnap

From Source Code to Binary

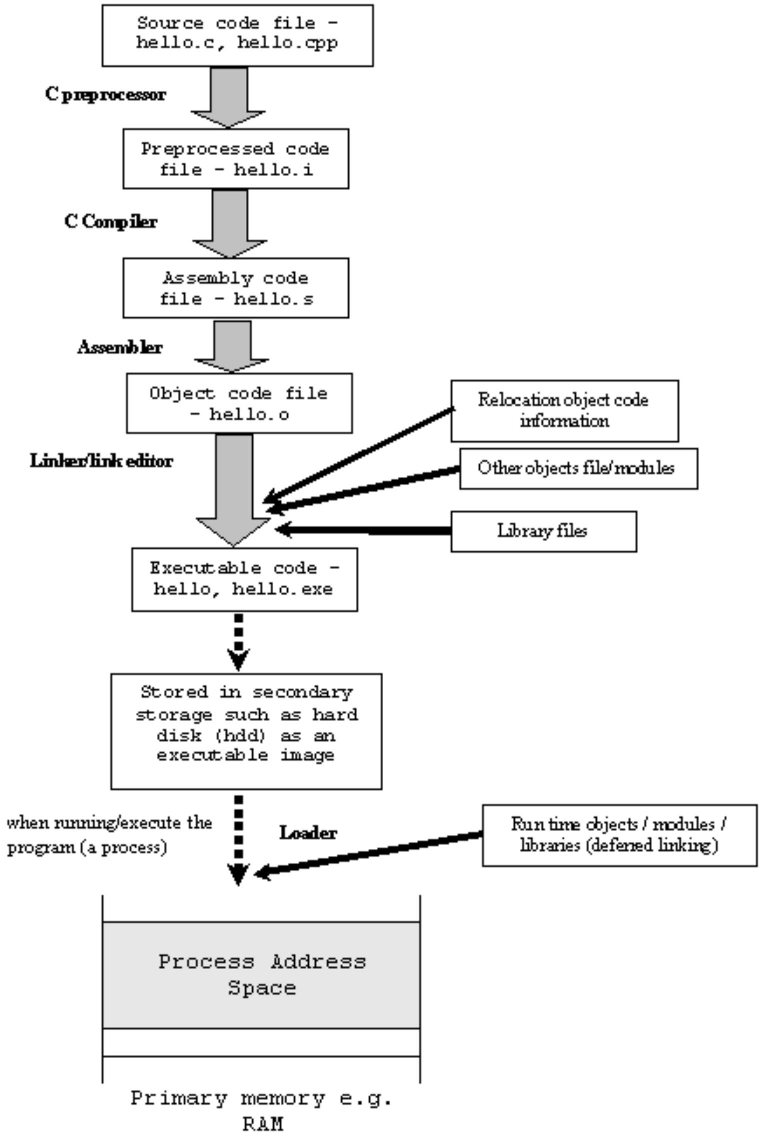
Universität Bamberg

- The compilation process
- Compiler, assembler, linker
- The ELF file format
- What comes before main?
- Tools
- Bare metal programming
- Writing and running the "Hello, world" example

Compiling code

Universität Bamberg

- Preprocessor
  - Expands #includes and macros
- Compiler
  - Generates assembler source code from C source code
- Assembler
  - Generates object code from assembler source code
- Linker
  - Combines (one or) multiple object files (+ libraries) to an executable file
- Loader
  - Loads executable file into main memory



Example: From .c to .o

Universität Bamberg

```c
#include <stdio.h>
static void display(int i, int *ptr);

int main(void)
{
	int x = 5;
	int *xptr = &x;
	printf("In main() program:\n");
	printf("x value is %d and is stored at address %p.\n", x, &x);
	printf("xptr pointer points to addr %p which holds a value %d.\n", xptr, *xptr);
	display(x, xptr);
	return 0;
}

void display(int y, int *yptr)
{
	char var[7] = "ABCDEF";
	printf("In display() function:\n");
	printf("y value is %d and is stored at address %p.\n", y, &y);
	printf("yptr pointer points to addr %p which holds a value %d.\n", yptr, *yptr);
}
```

Example: From .c to .o (2)

compiler: translates C source code file to object file `testprog.o`

```
$ gcc -c testprog1.c

$ ls -l
total 8
-rw-r--r-- 1 me me  629 Mar 22 13:15 testprog1.c
-rw-r--r-- 1 me me 1412 Mar 22 13:16 testprog1.o
```

check the type of the object file with `file`

```
$ file testprog1.o
testprog1.o: ELF 32-bit LSB relocatable, intel 80386, version 1 (SYSV), not
stripped
```

look at the contents of the object file with `hexdump`

```
$ hexdump -C testprog1.o
00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  01 00 03 00 01 00 00 00  00 00 00 00 00 00 00 00  |................|
00000020  84 02 00 00 00 00 00 00  34 00 00 00 00 00 28 00  |……..4.....(.|
00000030  0b 00 08 00 8d 4c 24 04  83 e4 f0 ff 71 fc 55 89  |.....L$.....q.U.|
[…]
```

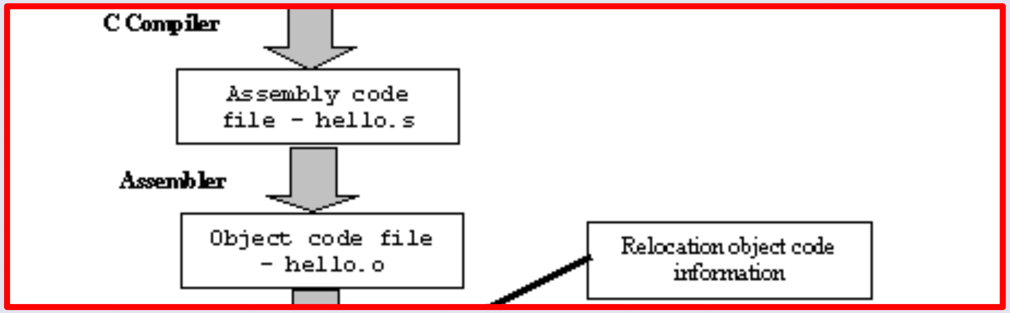
hexadecimal byte values

printable ASCII char's

Wait a moment…

Where is our assembler source?

Only generated internally as temporary file!



Compiler option "-S" explicitly generates a .s file:

```
$ gcc -S testprog1.c
$ ls -1
testprog1.c
testprog1.o
testprog1.s
```

```
        .file   "testprog1.c"
        .section        .rodata
.LC0:
        .string "In main() program:"
        .align 4
.LC1:
        .string "x value is %d and is stored at address %p.\n"
        .align 4
.LC2:
        .string "xptr pointer points to address %p which holds a value of %d.\n"
        .text
.globl main
        .type   main, @function
main:
        leal    4(%esp), %ecx
        andl    $-16, %esp
        pushl   -4(%ecx)
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ecx
        subl    $36, %esp
        movl    $5, -12(%ebp)
        leal    -12(%ebp), %eax
```

ELF internals (1)


```
$ file testprog1.o
testprog1.o: ELF 32-bit LSB relocatable, intel 80386, version 1 (SYSV),
not stripped
```

ELF:
Executable and Linking Format

```
$ man elf
[...]
The ELF header is described by the type Elf32_Ehdr or Elf64_Ehdr:
#define EI_NIDENT 16
typedef struct {
    unsigned char e_ident[EI_NIDENT];
    uint16_t      e_type;
    uint16_t      e_machine;
    uint32_t      e_version;
    ElfN_Addr     e_entry;
    ElfN_Off      e_phoff;
    ElfN_Off      e_shoff;
    uint32_t      e_flags;
    uint16_t      e_ehsize;
    uint16_t      e_phentsize;
    uint16_t      e_phnum;
    uint16_t      e_shentsize;
    uint16_t      e_shnum;
    uint16_t      e_shstrndx;
} ElfN_Ehdr;
```

ELF internals (2)

Universität Bamberg

```
$ hexdump -C testprog1.o
00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  01 00 03 00 01 00 00 00  00 00 00 00 00 00 00 00  |................|
00000020  84 02 00 00 00 00 00 00  34 00 00 00 00 00 28 00  |........4.....(.|
00000030  0b 00 08 00 8d 4c 24 04  83 e4 f0 ff 71 fc 55 89  |.....L$.....q.U.|
[…]
```

```
00000000  7f 45 4c 46 ; ELF „Magic“
00000004  01
00000005  01
00000006  01
00000007  00 00 00 00 00 00 00 00 00
```

/usr/include/elf.h:

```
#define ELFMAG        "\177ELF" // \177: octal
#define EI_CLASS      4 /* File class byte index
#define ELFCLASSNONE  0 /* Invalid class */
#define ELFCLASS32    1 /* 32-bit objects */
#define ELFCLASS64    2 /* 64-bit objects */
#define ELFCLASSNUM   3
```

```
#define EI_NIDENT 16
typedef struct {
    unsigned char e_ident[EI_NIDENT];
    uint16_t      e_type;
    uint16_t      e_machine;
    uint32_t      e_version;
    ElfN_Addr     e_entry;
    ElfN_Off      e_phoff;
    ElfN_Off      e_shoff;
    uint32_t      e_flags;
    uint16_t      e_ehsize;
    uint16_t      e_phentsize;
    uint16_t      e_phnum;
    uint16_t      e_shentsize;
    uint16_t      e_shnum;
    uint16_t      e_shstrndx;
} ElfN_Ehdr;
```

Endianness

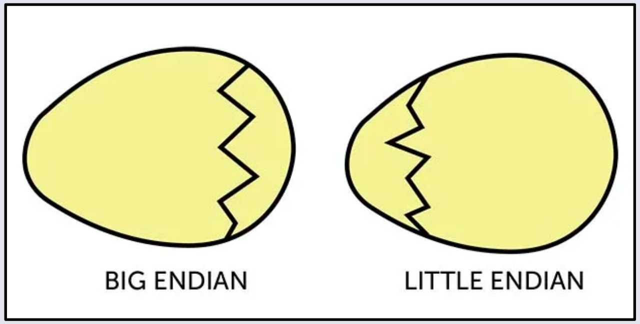
Universität Bamberg

/usr/include/elf.h:

```
#define EI_DATA         5     /* Data encoding byte index */
#define ELFDATANONE     0     /* Invalid data encoding */
#define ELFDATA2LSB     1     /* 2's complement, little endian */
#define ELFDATA2MSB     2     /* 2's complement, big endian */
```

Endianness or byte order

- Order in which the bytes of a data type using more than 8 bit are stored in memory
- Example:
  Decimal number 123456789 = hexadecimal **0x075BCD15**



Inspiration: Swift's "Gulliver's Travels" [4]

Little endian: least significant byte first (at lowest address)

00000000 **15**
00000001 **CD**
00000002 **5B**
00000003 **07**

Big Endian: most significant byte first:

00000000 **07**
00000001 **5B**
00000002 **CD**
00000003 **15**

Dissect an ELF with readelf

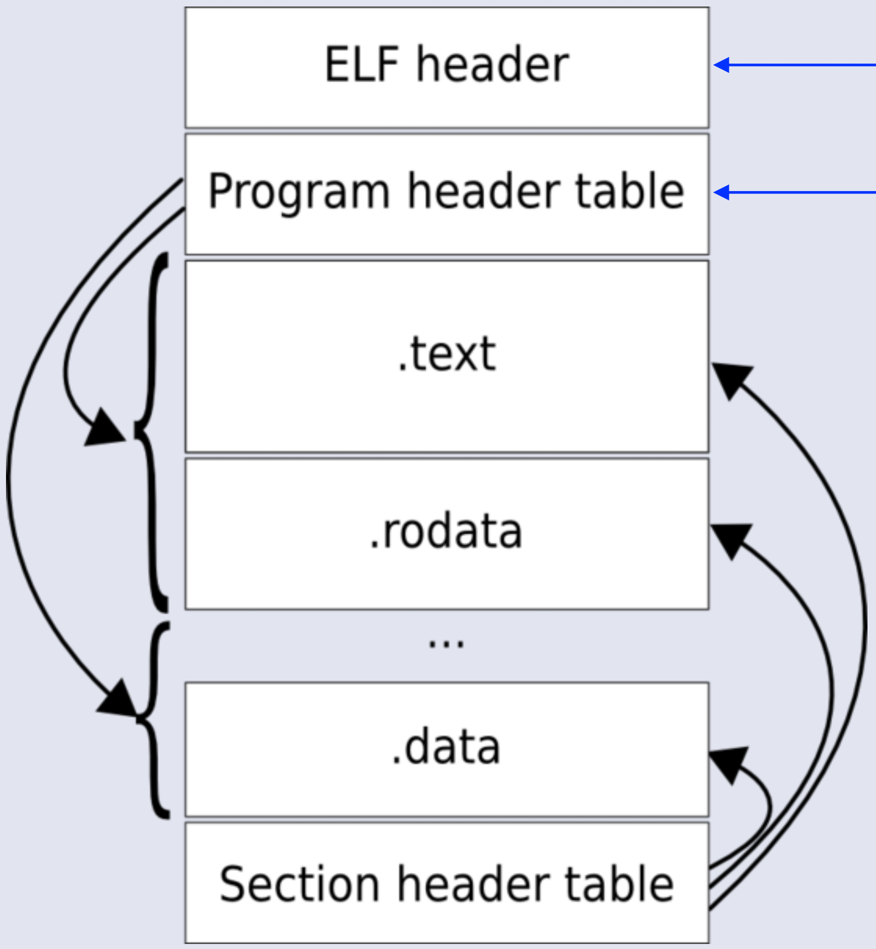

```
$ readelf testprog1.o
Usage: readelf <option(s)> elf-file(s)
 Display information about the contents of ELF format files
 Options are:
  -a --all               Equivalent to: -h -l -S -s -r -d -V -A -I
  -h --file-header       Display the ELF file header
[...]

$ readelf –h testprog1.o
```

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          644 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           40 (bytes)
  Number of section headers:         11
  Section header string table index: 8
```

ELF file structure

Universität Bamberg

The ELF file format is standardized (see [1] and [2])



Header with general information (see previous slides)

Information on the ELF file segments

Table of section headers with detailed information on each segment

ELF sections

Universität Bamberg

```
$ readelf -S testprog1.o
There are 11 section headers, starting at offset 0x284:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 0000db 00  AX  0   0  4
  [ 2] .rel.text         REL             00000000 000524 000060 08      9   1  4
  [ 3] .data             PROGBITS        00000000 000110 000000 00  WA  0   0  4
  [ 4] .bss              NOBITS          00000000 000110 000000 00  WA  0   0  4
  [ 5] .rodata           PROGBITS        00000000 000110 000102 00   A  0   0  4
  [ 6] .comment          PROGBITS        00000000 000212 00001f 00      0   0  1
  [ 7] .note.GNU-stack   PROGBITS        00000000 000231 000000 00      0   0  1
  [ 8] .shstrtab         STRTAB          00000000 000231 000051 00      0   0  1
  [ 9] .symtab           SYMTAB          00000000 00043c 0000c0 10     10   9  4
  [10] .strtab           STRTAB          00000000 0004fc 000026 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

ELF sections: what goes where?

[illegible]

```c
const int a = 42;
int b = 23;
int c;

int main(void) {
  c = a + b;
  return c;
}
```

What is where in the object file?

$ [illegible]
$ [illegible]

```
There are 11 section headers, starting at offset 0xd8:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 00002c 00  AX  0   0  4
  [ 2] .rel.text         REL             00000000 000364 000020 08      9   1  4
  [ 3] .data             PROGBITS        00000000 000060 000004 00  WA  0   0  4
  [ 4] .bss              NOBITS          00000000 000064 000000 00  WA  0   0  4
  [ 5] .rodata           PROGBITS        00000000 000064 000004 00   A  0   0  4
```

[illegible]	[illegible]
text (.text)	Machine code (instructions) and entry point (address)
read only data (.rodata)	initialized constants
read/write data (.data)	initialized variables
Base Storage Segment (.bss)	uninitialised data
Symbols (.symtab)	addresses for symbolic names

Both .data and .rodata hold a variable using 4 bytes each = sizeof(int)

ELF section details

Universität Bamberg

Offsets in ELF .o file:
???????????

```
There are 11 section headers, starting at offset 0xd8:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 00002c 00  AX  0   0  4
  [ 2] .rel.text         REL             00000000 000364 000020 08      9   1  4
  [ 3] .data             PROGBITS        00000000 000060 000004 00  WA  0   0  4
  [ 4] .bss              NOBITS          00000000 000064 000000 00  WA  0   0  4
  [ 5] .rodata           PROGBITS        00000000 000064 000004 00   A  0   0  4
```

```
00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  01 00 03 00 01 00 00 00  00 00 00 00 00 00 00 00  |................|
00000020  d8 00 00 00 00 00 00 00  34 00 00 00 00 00 28 00  |........4.....(.|
00000030  0b 00 08 00 8d 4c 24 04  83 e4 f0 ff 71 fc 55 89  |.....L$.....q.U.|
00000040  e5 51 8b 15 00 00 00 00  a1 00 00 00 00 8d 04 02  |.Q..............|
00000050  a3 00 00 00 00 a1 00 00  00 00 59 5d 8d 61 fc c3  |..........Y].a..|
00000060  17 00 00 00 2a 00 00 00  00 47 43 43 3a 20 28 44  |....*....GCC: (D|
00000070  65 62 69 61 6e 20 34 2e  33 2e 32 2d 31 2e 31 29  |ebian 4.3.2-1.1)|
```

?????????????

0x60: 17 00 00 00 => 0x00000017 ???????

0x64: 2a 00 00 00 => 0x0000002a ??????

ELF sections and assembler code

Universität Bamberg

In the assembler code:

```
$ ?????????????
$ ??????????
```

```
        .file    "foo.c"
.globl a
        .section .rodata
        .align 4
        .type    a, @object
        .size    a, 4
a:
        .long    42
.globl b
        .data
        .align 4
        .type    b, @object
        .size    b, 4
b:
        .long    23
```

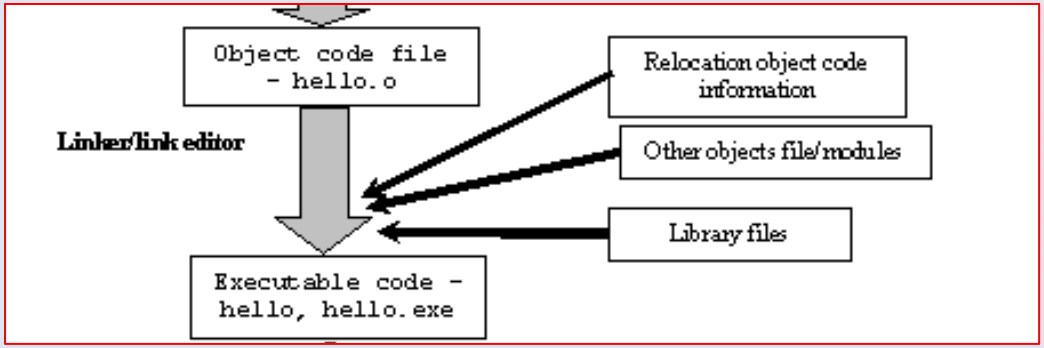
```
        .text
.globl main
        .type    main, @function
main:
        leal     4(%esp), %ecx
        andl     $-16, %esp
        pushl    -4(%ecx)
        pushl    %ebp
        movl     %esp, %ebp
        pushl    %ecx
        movl     a, %edx
        movl     b, %eax
        leal     (%edx,%eax), %eax
        movl     %eax, c
        movl     c, %eax
        popl     %ecx
        popl     %ebp
        leal     -4(%ecx), %esp
        ret
        .size    main, .-main
        .comm    c,4,4
        .ident   "GCC: (Debian 4.3.2-1.1) 4.3.2"
        .section .note.GNU-stack,"",@progbits
```

```
??????????

const int a = 42;
int b = 23;
int c;

int main(void) {
  c = a + b;
  return c;
}
```

Object files – and then?

Universität Bamberg

.o object files cannot be executed directly!



- Important parts of an executable file are missing
  - crt0 – startup code
  - initialization
    - variables in .bss need to be initialized (to 0)
    - for C++: calling of constructors
  - jump to "main" function and parameter passing (argc, argv, envp)
  - libraries, e.g. libc (C standard library), have to be added
- **Addresses of variables and functions are not resolved**
  - One of the main tasks of the linker

Symbols and addresses in object files

Addresses of functions and (global) variables have to be known

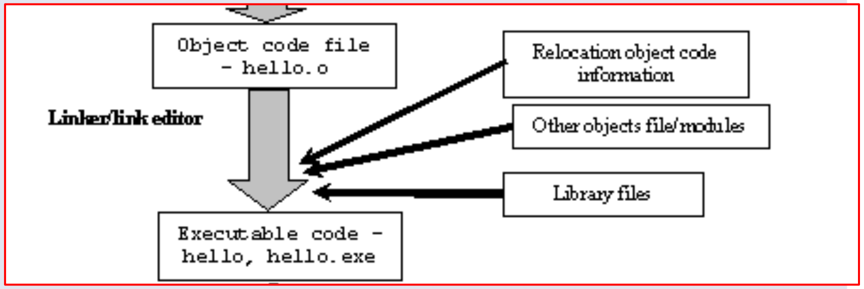
- Symbol table: assigns addresses to symbolic names for functions and variables:

```
$ readelf -s foo.o

Symbol table '.symtab' contains 12 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 FILE    LOCAL  DEFAULT  ABS foo.c
     2: 00000000     0 SECTION LOCAL  DEFAULT    1
     3: 00000000     0 SECTION LOCAL  DEFAULT    3
     4: 00000000     0 SECTION LOCAL  DEFAULT    4
     5: 00000000     0 SECTION LOCAL  DEFAULT    5
     6: 00000000     0 SECTION LOCAL  DEFAULT    7
     7: 00000000     0 SECTION LOCAL  DEFAULT    6
     8: 00000000     4 OBJECT  GLOBAL DEFAULT    5 a
     9: 00000000     4 OBJECT  GLOBAL DEFAULT    3 b
    10: 00000000    44 FUNC    GLOBAL DEFAULT    1 main
    11: 00000004     4 OBJECT  GLOBAL DEFAULT  COM c
```

- Addresses are set to 0 in the object file

Linker and relocations

Universität Bamberg



Linker receives a list of all object files required to build a program

- **.o** files (generated by the compiler or assembler): *object files*
  - can also be generated by C++, Fortran, ... compilers!
- **.a** files: "*archives*" of object files
  - *static libraries*
- **.so** files: "*shared object*"s
  - *dynamic libraries* (Windows: DLL "Dynamic Loadable Library")

Linker assigns text and data segments of the single .o files to parts of the address space for loading by the OS

- It resolves *references* to *symbols* (variables, functions)
- Controlled by a *linker script* (configuration file)

Linker and symbol resolution

Universität Bamberg

Symbol table of the linked program

- Also contains symbols of all linked libraries – a bit confusing…

```
$ gcc –o foo foo.o

$ readelf -s foo

Symbol table '.symtab' cont
Num:    Value  Size Type
[…]
 64: 08048460     4 OBJECT  GLOBAL DEFAULT   15 a
[…]
 69: 0804956c     4 OBJECT  GLOBAL DEFAULT   23 b
[…]
 73: 08048374    44 FUNC    GLOBAL DEFAULT   13 main
 74: 08048254     0 FUNC    GLOBAL DEFAULT   11 _init
 75: 08049578     4 OBJECT  GLOBAL DEFAULT   24 c
```

Linker and symbol resolution (2)

Universität Bamberg

Symbol table of the linked program

- Alternative: use "nm" ("names") from GNU binutils

```
$ gcc –o foo foo.o
$ nm foo
[…]
08048254 T _init
080482c0 T _start
08048460 R a
0804956c D b
08049578 B c
[…]
08048374 T main
```

?	????
T	.text
R	.rodata
D	.data
B	.bss

????????	?????????
text (.text)	Machine code (instructions) and entry point (address)
read only data (.rodata)	initialized constants
read/write data (.data)	initialized variables
Base Storage Segment (.bss)	uninitialised data
Symbols (.symtab)	addresses for symbolic names

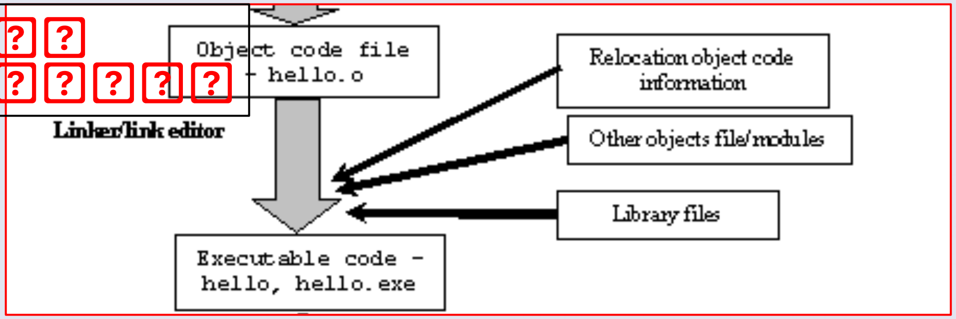
?????

```
$ readelf -s foo
Symbol table '.symtab' contains 76 entries:
Num:    Value  Size Type    Bind   Vis      Ndx Name
[…]
 64: 08048460     4 OBJECT  GLOBAL DEFAULT   15 a
 69: 0804956c     4 OBJECT  GLOBAL DEFAULT   23 b
 73: 08048374    44 FUNC    GLOBAL DEFAULT   13 main
 74: 08048254     0 FUNC    GLOBAL DEFAULT   11 _init
 75: 08049578     4 OBJECT  GLOBAL DEFAULT   24 c
```

✓

Linker and symbol resolution: .o files

????????
???????????

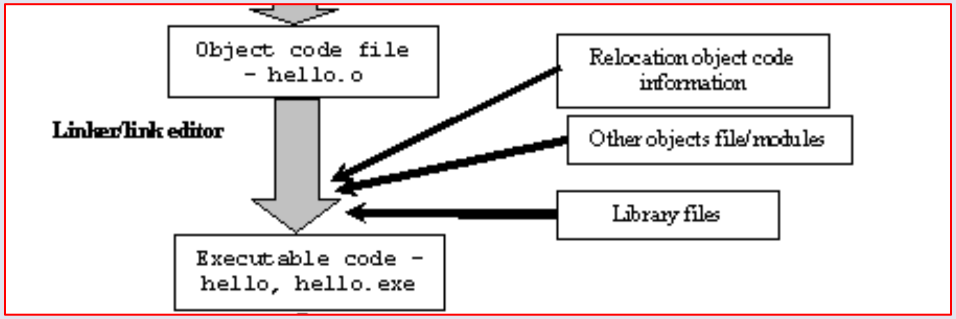



```
$ objdump -d foo.o

foo.o:     file format elf32-i386

Disassembly of section .text:

00000000 <main>:
   0:   8d 4c 24 04             lea    0x4(%esp),%ecx
   4:   83 e4 f0                and    $0xfffffff0,%esp
   7:   ff 71 fc                pushl  -0x4(%ecx)
   a:   55                      push   %ebp
   b:   89 e5                   mov    %esp,%ebp
   d:   51                      push   %ecx
   e:   8b 15 00 00 00 00       mov    0x0,%edx
  14:   a1 00 00 00 00          mov    0x0,%eax
  19:   8d 04 02                lea    (%edx,%eax,1),%eax
  1c:   a3 00 00 00 00          mov    %eax,0x0
  21:   a1 00 00 00 00          mov    0x0,%eax
  26:   59                      pop    %ecx
  27:   5d                      pop    %ebp
  28:   8d 61 fc                lea    -0x4(%ecx),%esp
  2b:   c3                      ret
```

??????????
??????????
??????????

Relocation table

- Contains information about the *relative location* of the related address in the text section for addresses initialized to "0" in the object file (*relocation offset*) and the address length (R_386_32 = 32 bit)




```
$ readelf -r foo.o

Relocation section '.rel.text' at offset 0x364 contains 4 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000010  00000801 R_386_32          00000000   a
00000015  00000901 R_386_32          00000000   b
0000001d  00000b01 R_386_32          00000004   c
00000022  00000b01 R_386_32          00000004   c
```

Linker and relocations: resolution


```
Relocation section '.rel.text' at offset 0x364 contains 4 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000010  00000801 R_386_32          00000000   a
00000015  00000901 R_386_32          00000000   b
0000001d  00000b01 R_386_32          00000004   c
00000022  00000b01 R_386_32          00000004   c
```

```
00000000 <main>:
   0:   8d 4c 24 04             lea    0x4(%esp),%ecx
   4:   83 e4 f0                and    $0xfffffff0,%esp
   7:   ff 71 fc                pushl  -0x4(%ecx)
   a:   55                      push   %ebp
   b:   89 e5                   mov    %esp,%ebp
   d:   51                      push   %ecx
   e:   8b 15 00 00 00 00       mov    0x0,%edx
  14:   a1 00 00 00 00          mov    0x0,%eax
  19:   8d 04 02                lea    (%edx,%eax,1),%eax
  1c:   a3 00 00 00 00          mov    %eax,0x0
  21:   a1 00 00 00 00          mov    0x0,%eax
  26:   59                      pop    %ecx
  27:   5d                      pop    %ebp
  28:   8d 61 fc                lea    -0x4(%ecx),%esp
  2b:   c3                      ret
```

Resolved resolutions in the executable

Universität Bamberg

```
$ objdump -d foo

foo:     file format elf32-i386

Disassembly of section .text:

08048374 <main>:
 8048374: 8d 4c 24 04             lea    0x4(%esp),%ecx
 8048378: 83 e4 f0                and    $0xfffffff0,%esp
 804837b: ff 71 fc                pushl  -0x4(%ecx)
 804837e: 55                      push   %ebp
 804837f: 89 e5                   mov    %esp,%ebp
 8048381: 51                      push   %ecx
 8048382: 8b 15 60 84 04 08       mov    0x8048460,%edx
 8048388: a1 6c 95 04 08          mov    0x804956c,%eax
 804838d: 8d 04 02                lea    (%edx,%eax,1),%eax
 8048390: a3 78 95 04 08          mov    %eax,0x8049578
 8048395: a1 78 95 04 08          mov    0x8049578,%eax
 804839a: 59                      pop    %ecx
 804839b: 5d                      pop    %ebp
 804839c: 8d 61 fc                lea    -0x4(%ecx),%esp
 804839f: c3                      ret
```

An ELF is not enough…

Universität Bamberg

- When running an application under an operating system, the *loader* takes care of interpreting the ELF file and copying its contents into memory as required
  - But… **we are writing the operating system :)**

- What does a processor do after power on (or reset)?
  - Fetch the very first instruction from a predefined address (called reset vector)
  - The processor starts up in the most privileged mode
- A regular PC has *firmware* in non-volatile memory (flash ROM)
  - BIOS or UEFI
- RISC-V is different
  - Regular: OpenSBI and u-boot – but we will start without firmware!

From ELF to binary

- How can we convert the ELF file to something the processor can execute?
  - The ELF headers and other meta information would confuse the processor
    - qemu can actually also load ELF files, but the real hardware cannot…
  - All information that is not executable code or data has to be removed

- Solution: the **objcopy** tool:

```
riscv64-unknown-elf-objcopy -O binary hello hello.bin
```

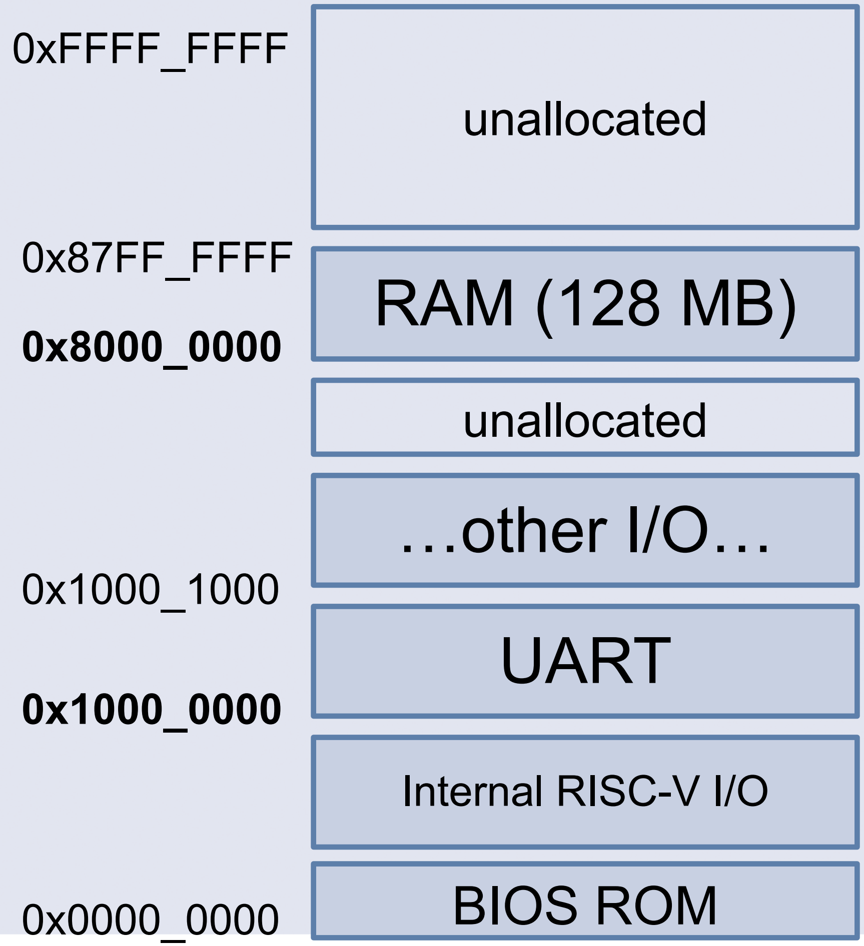
Tools of the trade

- **Compiler**
  - Translates C (or C++, …) source code into object files
  - We can use the GNU C compiler (GCC) or clang+LLVM
  - However, we need a **cross compiler** capable of generating instructions for a RISC-V processor!
- **Linker**
  - Links all object files of a program into a single executable
  - Part of GNU binutils (ld) – also a **cross-linker**
- **Make** and **Makefile**
  - Makefiles define rules to build a more complex target from multiple pieces
- **Loader**
  - Either some firmware (e.g. OpenSBI) or… no firmware at all
- **Emulator**
  - qemu – open source emulator for many different processor architectures
- **Debugger**
  - Connect to emulator and analyze code and data at runtime

Hardware is hard…

Universität Bamberg

- The *memory map* of a processor contains different *regions*
  - Read-only (ROM) and read-write (RAM) memory
  - I/O devices
- For our RISC-V emulated in qemu, the memory map looks like shown on the right



- Important for now:
  - UART at **0x1000_0000**
  - RAM at **0x8000_0000**
- qemu loads our hello.bin into memory at **0x8000_0000**

Our first (bare metal) program `hello.c`

Universität Bamberg

It prints "Hello, world!" – what else?

```
#include <stdio.h>

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

- **Problems running on bare metal:**
  - No libc – so no printf and no stdio.h header file
  - Startup code required to prepare for executing main
  - No operating system – no device drivers to actually output text

Next try…

Universität Bamberg

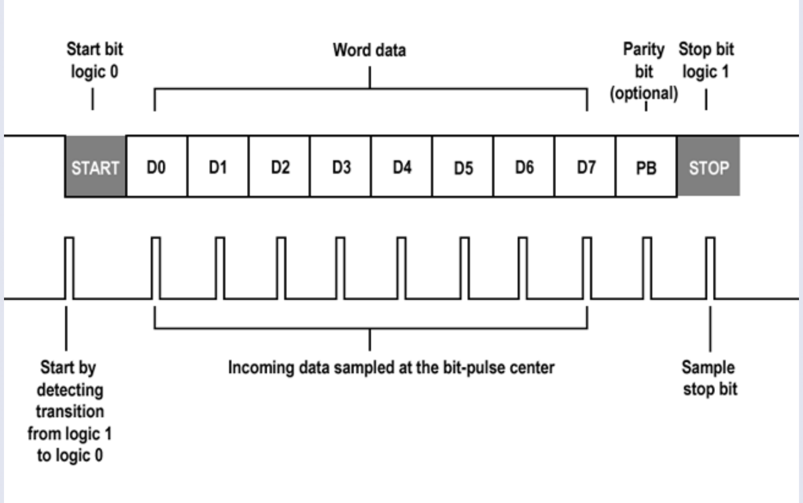
"Hello, world!" – `hello.c` version 2

```
void printstring(char *s) {
    …
}

int main(void) {
    printstring("Hello, world!\n");
    return 0;
}
```

- We have to write our own function to output text
  - Here called "`printstring`", this is similar to `puts`, since `printf` has a much more complex parameter handling
  - How can we implement this function?

How can we output text?

- Often there are multiple options (hardware interfaces)
  - Video output (frame buffer and display e.g. via HDMI or VGA)
  - Small LCD screen (connected via e.g. SPI or I2C)
  - Serial line (UART – universal asynchronous receiver/ transmitter) – **easiest way!**
- UARTs are ICs that transmit characters in **bit serial mode**
  - One bit after the other over a **three wire connection** (transmit, receive and ground signal line)



- For real UARTs, you need to configure parameters like speed, transmit and receive parameters (e.g. number of stop bits) etc.
- We can ignore this in qemu – real hardware requires this setup!

The 16550 UART

- The 16550 UART (used in the IBM PC since 1981 as 8250) is one of the standard UART "chips" available
  - Today, it is no longer a separate IC, but a part of the system-on-chip (SoC) silicon chip
- The data sheet [5] contains a list of the chip registers and a description of their functions
- Registers are often accessible as consecutive bytes starting at the UART base address
- Registers with DLAB=1 are only accessible when LCR bit 7 = 1

Offset	DLAB	R/W	Function
0	0	R	Receive buffer reg. RBR
0	**0**	**W**	**Transmit hold reg. THR**
0	1	R/W	Divisor latch (LSB)
1	0	R/W	Interrupt enable reg. IER
1	1	R/W	Divisor latch (MSB)
2	X	R	Interrupt identific. reg. IIR
2	X	W	FIFO control reg. FCR
3	X	R/W	Line control reg. LCR
4	X	R/W	Modem control reg. MCR
5	**X**	**R/W**	**Line status reg. LSR**
6	X	R/W	Modem status reg. MSR
7	X	R/W	Scratch register

The 16550 UART

Universität Bamberg

- Registers are accessible as consecutive bytes starting at the UART base address
  - For qemu: UART base = 0x1000_0000
- So RBR is at address 0x1000_0000, IER at 0x1000_0001 etc.
- Read and write operations to an address can access different functionality: read 0x1000_0000=RBR, write 0x1000_0000=THR
- Most registers combine bits for different functions
- Example: Line status register LSR
  - Transmit/Receive status plus error conditions
- Important for us now:
  - LSR at 0x1000_0005
    Bit 5 "THR empty"

LSR bit	Function
0	Data available
1	Overrun error
2	Parity error
3	Framing error
4	Break signal received
5	THR empty
6	THR empty & line idle
7	Erroneous data in FIFO

Polling a UART

Universität Bamberg

- Input/output (I/O) devices can be run in two modes
  - **Interrupt-driven:** processor receives a signal (*interrupt*) as soon as data arrives or can be sent using the device
  - **Polling:** processor has to check continually (in a loop) if the I/O device is ready for the next piece of data
- How can our software "talk" to the UART?
  - *Registers* of the UART are available at predefined addresses
  - Reading/writing the registers result in actions of the UART
- Writing a character to a UART in polling mode looks like this (pseudocode):

```
while not (uart_transmitter_ready) { wait… }
write character to transmit register
```

Mapping the UART in software

Universität Bamberg

We define a structure type that describes the registers:

```
typedef char uint8_t; // defined for convenience

struct uart {
   uint8_t THR; // transmit hold register (offset 0)
   uint8_t IER;
   uint8_t IIR;
   uint8_t LCR;
   uint8_t MCR;
   uint8_t LSR; // line status register (offset 5)
};

volatile struct uart* uart0 = (volatile struct uart*)0x10000000;
```

- The **uart** struct maps the registers to memory addresses
- The start address of the struct is **initialized** to 0x1000_0000;
  - The first element (THR) is at relative offset 0 (0x1000_0000)
  - …and the LSR at relative offset 5 (0x1000_0005)

Polling the UART transmitter

Universität Bamberg

```
struct uart {
    …  // as before
};

volatile struct uart* uart0 = (volatile struct uart*)0x10000000;

void putachar(char c) {
    while ((uart0->LSR & (1<<5)) == 0)
        ;              // do nothing - wait until bit 5 of LSR = 1
    uart0->THR = c;    // then write the character
}

void printstring(char *s) {
    while (*s) {       // as long as the character is not null
        putachar(*s);  // output the character
        s++;           // and progress to the next character
    }
}
```

What about the "volatile"?

Universität Bamberg

Why is the "volatile" qualifier required here?

```
volatile struct uart* uart0 =
     (volatile struct uart*)0x10000000;
```

- Compilers are very eager to optimize away "unused" code
  - Bytes written to THR register are never read by the program
  - Bytes read from LSR are never written by the program
- The compiler cannot know that the addresses of our uart struct are not simple memory (RAM), but have *side effects* – controlling the UART
- Thus, volatile tells the compiler not to optimize away accesses to these variables
  - Volatile is one of the most misunderstood features of C [6]

Nitty gritty startup code: `boot.S`

Universität Bamberg

- A few assembler instructions are required to get to main
  - Initialize the **stack pointer SP**
  - Jump to main
  - Finally, an endless loop if main returns

```
        .section .text
        .global _entry
_entry:
        la   sp, stack0    // stack0 is a 4096 byte array
        li   a0, 4096      //
        add  sp, sp, a0    // set sp to end of stack

        jal  main          // subroutine call to main
loop:
        j    loop          // endless loop if main returns
```

So we need a stack in `hello.c`

```
__attribute__ ((aligned (16))) char stack0[4096];

struct uart {
    …
};

// …and the rest…
```

- The stack is simply a region in memory, represented by a C array
  - Stores return addresses, local variables, etc.
  - Required for function calls in C
- Here, stack0 is simply a 4096 byte array
  - Aligned to a multiple of 16 bytes (required by RISC-V)
- `boot.S` sets the stack pointer to the first byte **after** that array
  - This works since the stack pointer is first decremented (-4) before data is written to the stack!

Tell the linker where things belong…

- The **linker script** defines where in memory the different parts of the executable program are to be located ("stolen" from xv6…)

```
OUTPUT_ARCH( "riscv" )
ENTRY( _entry )
SECTIONS
{
  /* ensure _entry is at 0x80000000
   * where qemu -kernel jumps.
   */
  . = 0x80000000;

  .text : {
    *(.text .text.*)
    . = ALIGN(0x1000);
    PROVIDE(etext = .);
  }
  /* continued on the right side */
```

```
  .rodata : {
  . = ALIGN(16);
    *(.rodata .rodata.*)
  }

 .data : {
    . = ALIGN(16);
   *(.data .data.*)
  }

  .bss : {
  . = ALIGN(16);
    *(.bss .bss.*)
  }

  PROVIDE(end = .);
}
```

Now build it all

Universität Bamberg

- Compile the C and the assembler source file
- Link both into an ELF executable "hello"
- Convert the ELF file into a binary `hello.bin` so that qemu can execute it
  - Here, as a shell script – we'll build a makefile next time!
- **-ffreestanding**: don't depend on libc (or crt0) – we have our own

```
#!/bin/bash
riscv64-unknown-elf-gcc -g -mcmodel=medany -mno-relax \
        -c boot.S
riscv64-unknown-elf-gcc -g -mcmodel=medany -mno-relax \
        -c hello.c
riscv64-unknown-elf-gcc -g -ffreestanding -fno-common \
        -nostdlib -mno-relax -mcmodel=medany \
        -Wl,-T hello.ld *.o -o hello
riscv64-unknown-elf-objcopy -O binary hello hello.bin
```

Run it!

Universität Bamberg

- Run in qemu… **YES!**

Hello_World — -zsh — 69×5

~/Uni/Bamberg/Lehre/2022-SS/OSE/software/Hello_World — -zsh

```
$ qemu-system-riscv64 -nographic -machine virt -smp 1 -bios none \
> -kernel ./hello.bin
Hello, world!
QEMU: Terminated
$
```

- Parameters to qemu:
  - **-nographic**: run without video output, UART output in terminal
  - **-machine virt**: specific machine type (with virtual I/O)
  - **-smp 1**: emulate a single-processor system
  - **-bios none**: run without a bios
  - **-kernel ./hello.bin**: use our hello.bin program as kernel, mapped to 0x8000_0000
- Exit qemu with the following combination: first Control-A, then press X

How can we map dual-use registers?

Universität Bamberg

Some regs have different names for R and W at the same offsets

- This can be handled by using C **unions** – two names for one element
- The correct use is the responsibility of the programmer (i.e., you can still write to RBR, but it would nevertheless end up in the THR register)

```
struct uart {
   union {
     uint8_t THR; // W = transmit hold register (offset 0)
     uint8_t RBR; // R = receive buffer register (also offset 0)
     uint8_t DLL; // RW = divisor latch low (off. 0 when DLAB=1)
   };
   union {
     uint8_t IER; // RW = interrupt enable reg. (offset 1)
     uint8_t DLH; // divisor latch high (offset 1 when DLAB=1)
   };
   union {
     uint8_t IIR;   // R = interrupt identif. reg. (offset 2)
     uint8_t FCR;   // W = FIFO control reg. (also offset 2)
   };
   // …rest as before
};
```

Conclusion

Universität Bamberg

- **We made it!**
  - Our first program runs bare metal on a RISC-V (emulator)
- Such a simple "Hello world" program can be useful to…
  - check if your compilation toolchain is working
  - check if your emulator works
  - check if your assumption about the interaction of hardware and software is correct
- In the next lecture, we will take a detailed look at the RISC-V processor architecture
  - …and then we will dive into the xv6 operating system

References

1. System V Application Binary Interface Edition 4.1 (1997-03-18)
2. ELF TIS Committee, *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification, Version 1.2*
   Online unter https://www.uclibc.org/docs/elf.pdf
3. John R. Levine, *Linkers and Loaders*, MKP 1999, ISBN 1-55860-496-0 – Online (beta) at https://www.iecc.com/linker/
4. Jonathan Swift, *Gulliver's Travels, or Travels into Several Remote Nations of the World*, 1726
5. UART 16550 data sheet
   https://www3.tuhh.de/osg/Lehre/SS21/V_BSB/doc/uart-16550d.pdf
6. Eric Eide and John Regehr, *Volatiles Are Miscompiled, and What to Do about It*, Proc. of EMSOFT 2008