# **Foundations of** Embedded Systems

## **Topic 04:** Low-Level C Programming and Tools for Embedded Systems

**(Video)**

(~60 minutes)

Version 0.2020

Phillip **Stanley-Marbell**

Department of Engineering, University of Cambridge

http://physcomp.eng.cam.ac.uk

UNIVERSITY OF CAMBRIDGE

F(E)

(10)

# **Intended Learning Outcomes** for Today

**By the end of this topic, you should be able to:**

❶ Enumerate **steps in compiling a program** from C to machine instructions

❷ Enumerate and understand the **tools involved in the process**

❸ Enumerate the **main components of a processor (core) implementation**

❹ Understand and enumerate **how assembly code gets executed** in processor

❺ **Design** your own **linker script files** and use tools like **`objdump`** and **`objcopy`**

# Next:

- We will look at **four different variants of a simple 4-line C program** to introduce the compilation and linking process

# Anatomy of Four C Programs: Program 1



```
1 int
2 main(void) {
3     return 0;
4 }
```

```
sh-coff-gcc -E init.S > init.i; sh-coff-as init.i -o init.o
sh-coff-gcc -DM32 -Wall  -gstabs3 -O0  -c simple.c
sh-coff-ld -Ttext 0x08004000 -TsuperH.ld -L../../../tools/tools-lib/superH -L
     ../../../sys/libOS/mOS -Map simple.map init.o simple.o  -o simple -lc -lgcc -
     lm -lmOS-superH
sh-coff-objcopy -O srec simple simple.sr
```

More on **linker script files** (an *input*) later

More on **map files** (an *output*) later

# Anatomy of Four C Programs: Disassembling Program 1



```
% sh-coff-objdump -d simple

simple:     file format coff-sh

Disassembly of section .text:

0000000008004000 <_text>:
 8004000:   c9 00           and #0,r0
 8004002:   40 0e           ldc r0,sr

...      150 lines deleted   ...

 80040ee:   00 09           nop
```

**Disassembled binary is 160 lines**

# Anatomy of Four C Programs: Program 2



```
1 int
2 noMain(void) {
3     return 0;
4 }
```

```
sh-coff-gcc -E init.S > init.i; sh-coff-as init.i -o init.o
sh-coff-gcc -DM32 -Wall  -gstabs3 -O0  -c simple.c
sh-coff-ld -Ttext 0x08004000 -TsuperH.ld -L../../../tools/tools-lib/superH -L
     ../../../sys/libOS/mOS -Map simple.map init.o simple.o  -o simple -lc -lgcc -
     lm -lmOS-superH
sh-coff-objcopy -O srec simple simple.sr
```

**Exactly the same compilation flags and steps**

# Anatomy of Four C Programs: Disassembling Program 2

```
% sh-coff-objdump -d simple

simple:     file format coff-sh

Disassembly of section .text:

0000000008004000 <_text>:
 8004000:   c9 00           and #0,r0
 8004002:   40 0e           ldc r0,sr
 8004004:   df 02           mov.l   8004010 <stack_addr>,r15    ! 0xc001000
 8004006:   d0 03           mov.l   8004014 <start_addr>,r0 ! 0x8004020 <_noMain>
 8004008:   40 0b           jsr @r0
 800400a:   00 09           nop
 800400c:   e4 01           mov #1,r4
 800400e:   c3 22           trapa   #34
```

**What will be the difference in binaries** between Program 1 and Program 2?

**Disassembled binary is just 38 lines.** Only difference is change of function name

# Anatomy of Four C Programs: Program 3

```c
#include <stdio.h>

int
main(void) {
    return 0;
}
```

Compared to Program 1:

```c
int
main(void) {
    return 0;
}
```

**What will be the difference in binaries** between Program 1 and Program 3**?**

**Absolutely no difference**

# Anatomy of Four C Programs: Program 4



```c
#include <stdio.h>

int
main(void) {
    printf("Hello, World\n");

    return 0;
}
```

**(Compared to Program 3:)**

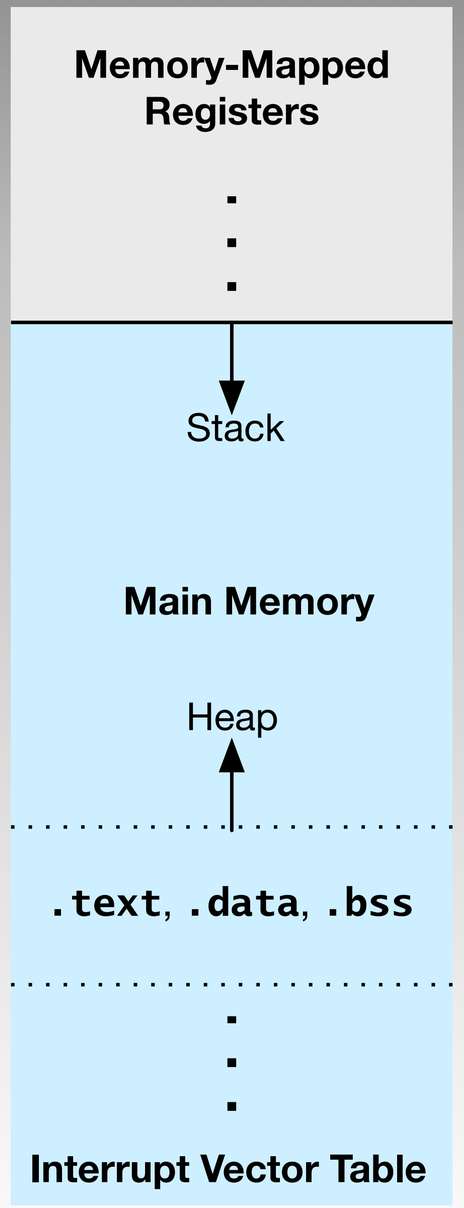
```c
#include <stdio.h>

int
main(void) {
    return 0;
}
```

**Disassembled binary will now be 4116 lines** (vs. 160 lines for Progs. 1 and 3)

# Memory Map and the Parts of a Binary: **Terminology**



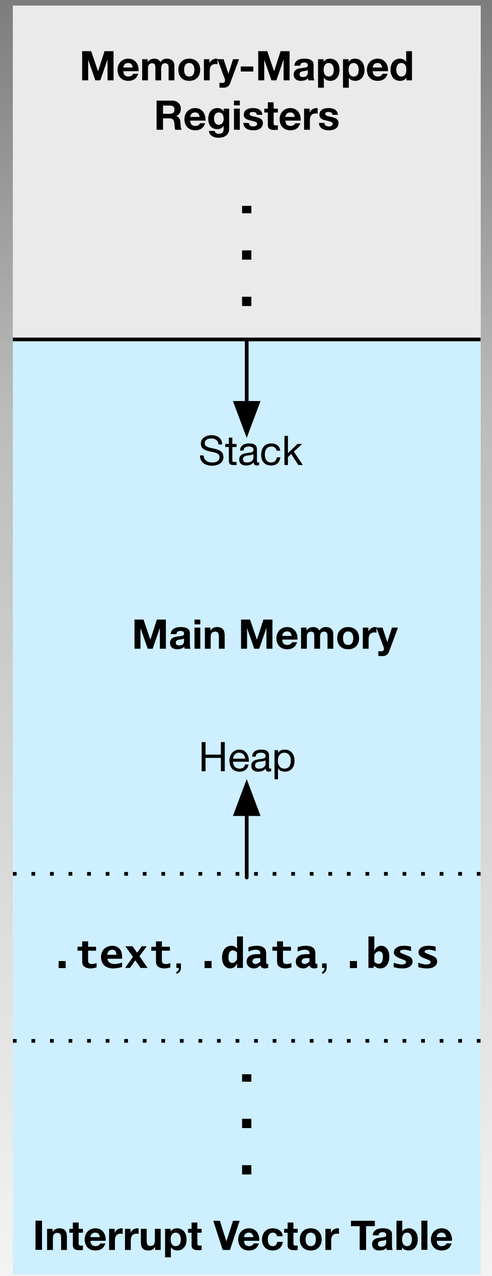



**Terminology**

- Memory map
- Memory-mapped registers
- Stack
- Heap
- **`.text`** section
- **`.data`** section
- **`.bss`** section

# How do we know what is where in memory/binary?

- Stack?
- Heap?
- `.text`?
- `.data`?
- `.bss`?




Like maps in the physical world, ***map files*** tell us what's where in the binary

# Map File:

- An **output file** generated by the linker which indicates what symbol names are in the final binary, which object files they came from, their sizes, and where in the binary they are located. (Map files often have the filename extension **`.map`**)

map | mæp |

noun
**1** a diagrammatic representation of an area of ...

# What's In a Binary?[1 of 3]: **Understanding Map Files**



```
1  int gInitializedInt = 42;
2  int gUninitializedInt;
3
4  int
5  noMain(void)
6  {
7      int i, localInt;
8
9      for (i = 0; i < 10; i++) {
10         localInt += gInitializedInt + gUninitializedInt;
11     }
12
13     return 0;
14 }
```

# What's In a Binary?[2 of 3]: **Understanding Map Files**



```
1  int gInitializedInt = 42;
2  int gUninitializedInt;
3
4  int
5  noMain(void)
6  {
7      int i, localInt;
```

**Map File** Lines 1 to 7:

```
1  Allocating common symbols
2  Common symbol       size              file
3
4  gUninitializedInt   0x4               simple.o
5
6  Memory Configuration
7
```

# What's In a Binary?3 of 3: **Understanding Map Files**



```
1  int gInitializedInt = 42;
2  int gUninitializedInt;
3
4  int
5  noMain(void)
```

**Map file**, lines 13 to 21:

```
13 Address of section .text set to 0x8004000
14
15 .text           0x0000000008004000       0x80
16                 0x0000000008004000                _text = .
17  *(.text)
18  .text          0x0000000008004000       0x20 init.o
19                 0x0000000008004018                __errno
20  .text          0x0000000008004020       0x60 simple.o
21                 0x0000000008004020                noMain
22                 0x0000000008004080                _etext = .
```

# Debugging Information:

- Typically part of (stored in) binary, but not loaded into processor for execution. **Contains information that allows you to correlate processor state to program source code**.

# What's In a Binary?[1 of 2]: **Debugging Information**



```
1  int gInitializedInt = 42;
2
3  int
4  noMain(void) {
5      int i, localInt;
6
7      for (i = 0; i < 10; i++) {
8          localInt += gInitializedInt + gUninitializedInt;
```

**"Stabs" debugging information (dumped)**, lines 1 to 10:

```
1  % sh-coff-objdump -G simple
2
3  simple:     file format coff-sh
4
5  Contents of .stab section:
6
7  Symnum n_type n_othr n_desc n_value  n_strx String
8
9  -1     HdrSym 0      31     0000000000000292 1
10 0      SO     0      2      0000000008004020 1      simple.c
```

# What's In a Binary?[2 of 2]: **Debugging Information**



```
1  int gInitializedInt = 42;
2
3  int
4  noMain(void) {
5      int i, localInt;
6
7      for (i = 0; i < 10; i++) {
8          localInt += gInitializedInt + gUninitializedInt;
```

**"Stabs" debugging information (dumped)**, lines 27 to 36:

```
27 17     FUN    0      0      0000000008004020 573    noMain:F(0,1)
28 18     SLINE  0      6      0000000000000000 0
29 19     SLINE  0      9      0000000000000006 0
30 20     SLINE  0      11     0000000000000012 0
31 21     SLINE  0      9      000000000000002a 0
32 22     SLINE  0      14     0000000000000044 0
33 23     SLINE  0      15     0000000000000048 0
34 24     LSYM   0      0      0000000000000000 587    i:(0,1)
35 25     LSYM   0      0      0000000000000004 595    localInt:(0,1)
36 26     LBRAC  0      0      0000000000000000 0
```

# What's In a Binary? 4 of 4: Understanding Map Files



```
1 int gInitializedInt = 42;
2 int gUninitializedInt;
3
4 int
5 noMain(void)
```

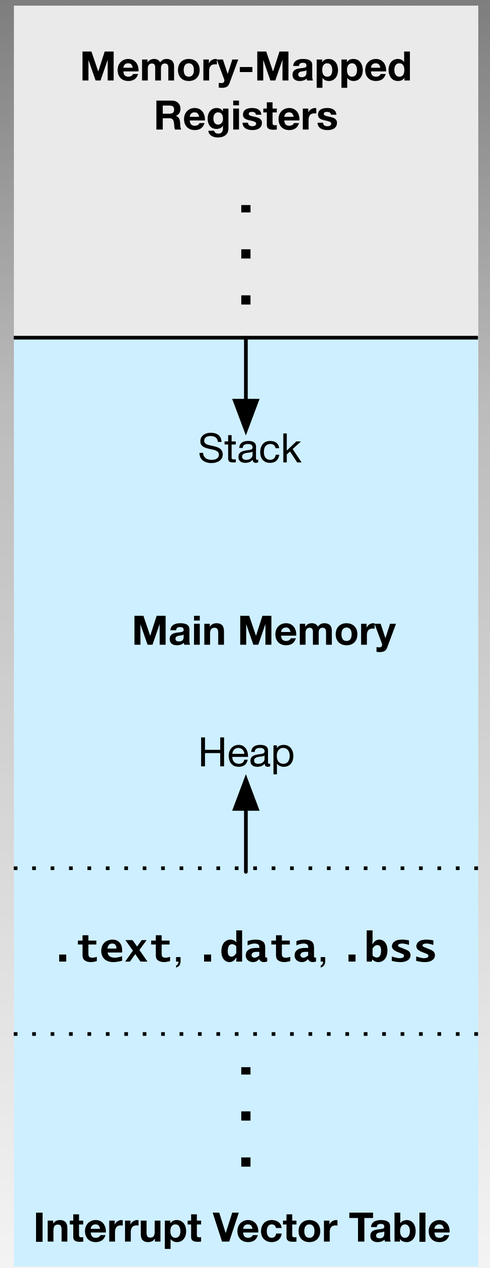
## Map File

```
LOAD init.o
LOAD simple.o
LOAD ../../../tools/tools-lib/superH/libc.a
LOAD ../../../tools/tools-lib/superH/libgcc.a
LOAD ../../../tools/tools-lib/superH/libm.a
LOAD ../../../sys/libOS/mOS/libmOS-superH.a
OUTPUT(simple coff-sh)

.stab           0x0000000008004094      0x180
 .stab          0x0000000008004094      0x180 simple.o
```

# How do we control what goes where in memory/binary?

- Stack?
- Heap?
- `.text`?
- `.data`?
- `.bss`?




***Linker command files*** tell the linker how to layout the contents of binaries

# Linker Script File (Linker Command File)

- An **input command file,** used by the linker, which tells the linker how to place code in the final linked binary.

command | kəˈmænd |

noun
an authoritative order: it's unlikely they'll obey your commands.
• Computing an instruction or signal that causes...

# Linker Script Files 1 of 2



```
OUTPUT_FORMAT("coff-sh")
OUTPUT_ARCH(sh)
SECTIONS
{
        .text . :
        {
                        _text = . ;
                        *(.text)
                        *(.strings)
                        _etext = . ;
        }
        .tors :
        {
                ___ctors = . ;
                *(.ctors)
                ___ctors_end = . ;
                ___dtors = . ;
                *(.dtors)
                ___dtors_end = . ;
        }
        11 lines deleted...
        .bss . :
        {
                        _bss = . ;
                        *(.bss)
                        *(COMMON)
                        _ebss = . ;
                        _end = . ;
        }
}
```

# Linker Script Files 2 of 2

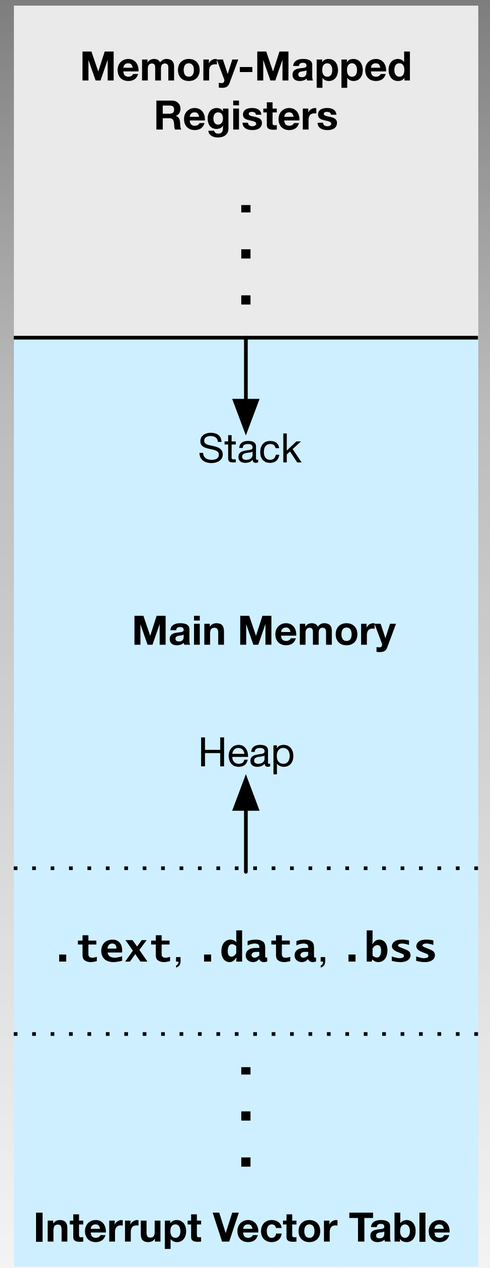


```
OUTPUT_FORMAT("coff-sh")
OUTPUT_ARCH(sh)
SECTIONS
{
        .text . :
        {
                        _text = . ;
                        *(.text)
                        *(.strings)
                        _etext = . ;
        }
        .tors :
        {
                ___ctors = . ;
                *(.ctors)
                ___ctors_end = . ;
```

Recap

# How do we control what goes where in memory/binary?

- Stack?
- Heap?
- `.text`?
- `.data`?
- `.bss`?




***Linker command files*** tell the linker how to layout the contents of binaries

# Next:

- We will look at **an extended example that combines C and assembly language** to implement **processor initialization** and a **relocated interrupt handler**

# Combining **C, Assembler,** and an **Interrupt Handler**





```
1  #include "print.h"
2
3  /*       8 GPRs + PR      */
4  unsigned char   REGSAVESTACK[36];
5  static void     hdlr_install(void);
6  volatile int    gFlag = 1;
7
8  int
9  main(void) {
10     hdlr_install();
11
12     print("\n\n\nStarting...\n\n\n");
13     while (gFlag) {
14     }
15     print("\n\n\nExiting, bye!\n\n\n");
16
17     return 0;
18 }
```

**and…**

```
19
20 void
21 intr_hdlr(void) {
22     gFlag = 0;
23
24     return;
25 }
26
```

# Combining **C, Assembler,** and an **Interrupt Handler**[2 of 9]



```
27  void
28  hdlr_install(void) {
29      extern  unsigned char   vec_stub_begin, vec_stub_end;
30      unsigned char *         dstptr = (unsigned char *)0x8000600;
31      unsigned char *         srcptr = &vec_stub_begin;
32
33      /*  Copy the vector instructions to vector base */
34      while (srcptr < &vec_stub_end)
35          *dstptr++ = *srcptr++;
36      }
37
38      return;
39  }
```

**Key Concept:** *explicitly referencing a specific memory address* (e.g., **`0x8000600`**) in C

# Combining **C, Assembler,** and an **Interrupt Handler**



```
1  #include "asm.h"
2
3      .global _vec_stub_begin
4      .global _vec_stub_end
5      .global _sleep
6      .global _atexit
7
8      .align  2
9  start:
10     /*  Clear Status Reg    */
11     AND #0, r0
12     LDC r0, sr
13
14     /*  Go !    */
15     MOVL    app_stack_addr, r15
16     MOVL    start_addr, r0
17     JSR @r0
18     NOP
```

**Note 1**

**Note 2**

# Combining **C, Assembler,** and an **Interrupt Handler** 4 of 9



```
28      .align  4
29  _vec_stub_begin:
30      /*      Save PR      */
Note 3  MOVL    savestack_addr, r0
32      ADD #36, r0
33      STS.L   pr, @-r0
34
35      /*      Save R8 - R15    */
36      BSR saveregs
37      NOP
38
39      /*  It's now safe to call    */
40      MOVL    intr_stack_addr, r15
```

**Recall**

```
27  void
28  hdlr_install(void) {
29      extern  unsigned char   vec_stub_begin, ve
30      unsigned char *         dstptr = (unsigned
31      unsigned char *         srcptr = &vec_stub
32
33      /*  Copy the vector instructions to vector
34      while (srcptr < &vec_stub_end) {
35          *dstptr++ = *srcptr++;
36      }
37
38      return;
39  }
```

**Key Concept:** *Symbols (e.g.,* **`vec_stub_begin`***)* in assembly/C/linker script file

# Combining **C, Assembler,** and an **Interrupt Handler** 5 of 9

```
41      MOVL    hdlr_addr, r0
42      JSR @r0
43      NOP
44
45      /*  Restore R8 - R15    */
46      BSR restoreregs
47      NOP
48
49      /*  Restore PR      */
Note 3  MOVL    savestack_addr, r0
51      ADD #32, r0
52      LDS.L   @r0+, pr
53
54      /*  Return from exception   */
55      RTE
56      NOP
```

# Combining **C, Assembler,** and an **Interrupt Handler** 6 of 9

```
59      /*                                      */
60      /*  SR.RB == 1. Save R8-R15. We store items in reverse  */
61      /*  so that we can use MOV.L Rm, @-Rn, rather than have */
62      /*  to incr the memory store address ourselves.     */
63      /*                                      */
64  saveregs:
65      /*  Addr of bottom of save area */
Note 3  MOVL    savestack_addr, r0
67
68      /*    Get addr _end_ of stack   */
69      ADD #32, r0
70
71      /*    Now store items bkwrds    */
72      MOVL    r15, @-r0
73      MOVL    r14, @-r0
74      MOVL    r13, @-r0
```

# Combining **C, Assembler,** and an **Interrupt Handler**





```
85      /*                                      */
86      /*          SR.RB == 1.  Restore R8-R15.        */
87      /*                                      */
88  restoreregs:
89      /*  Addr of bottom of save area */
Note 3  MOVL    savestack_addr, r0
91
92      /*      Pop into approp. reg    */
93      MOVL    @r0+, r8
94      MOVL    @r0+, r9
95      MOVL    @r0+, r10
96      MOVL    @r0+, r11
97      MOVL    @r0+, r12
98      MOVL    @r0+, r13
```

# Combining **C, Assembler,** and an **Interrupt Handler**





```
104
105         .align  4
106         hdlr_addr:
107         .long   _intr_hdlr
108
109         .align  4
Note 3!!!   savestack_addr:
111         .long   _REGSAVESTACK
112
113         .align  4
114         intr_stack_addr:
115         .long   (0x8000000 + (1 << 16))
116 _vec_stub_end:
```

**Key Concept: array `REGSAVESTACK[]` in the C code provides a place to stash saved state**

# Combining **C, Assembler,** and an **Interrupt Handler**





```
125
126     .align  2
127     /*  Stack is 32k above us    */
128 app_stack_addr:
129     .long   (0x8000000 + (1 << 15))
130
131     .align  2
132 start_addr:
133     .long   _main
```

**Key Concept:** `intr_hdlr()` and `main()` have separate 32k stacks. **32k big enough?** KL03?

# Further Reading

**Best next step: Get some practice**

- Like learning to swim, you can't learn all you need from a textbook

**A basic C programming tutorial you might find useful:**

https://www.edx.org/course/programming-in-c-getting-started

**Test your understanding:**

- Complete these online self-assessments on https://f-of-e.org/

https://f-of-e.org/chapter-04/#exercises

# Things to Do

❶ Complete a **“muddiest point”** 2-question survey using this link