Modular Coding/Linking

Computer Engineering 2

CT Team: A. Gieriet, J. Gruber, R. Gübeli, M. Meli, M. Rosenthal,
A. Rüst, J. Scheier, M. Thaler

Motivation

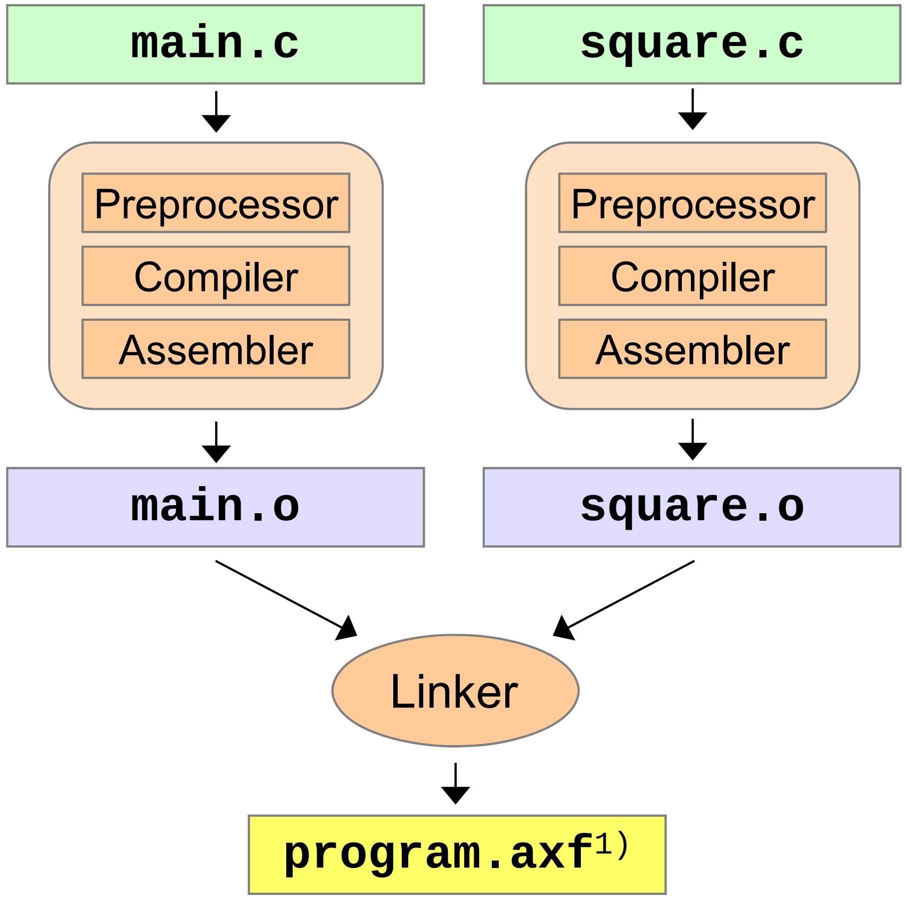
- **From source code to executable program**

main.c

```
...

uint32_t square(uint32_t v);

int main(void)
{
    while(1) {
        LED = square(DIPSW);
    }
}
```

square.c

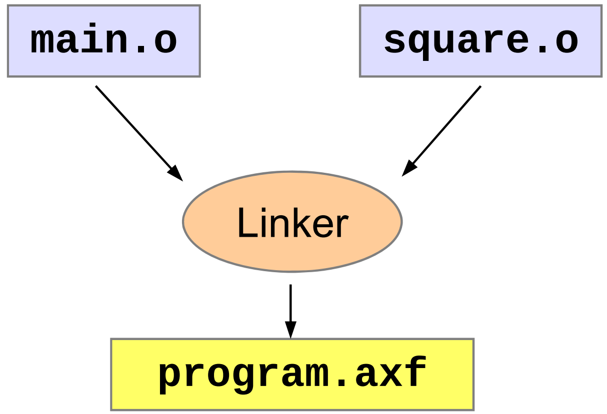
```
...

uint32_t square(uint32_t v)
{
    return v * v;
}
```



1) AXF file extension indicates ARM Executable File Format - other environments may have other executable formats

Agenda

- **Modular programming**
  - Why modular programming
  - Some guidelines for designing modular programs
- **From source code to the executable program**
  - Source code anatomy
  - Linker
    - Merging code sections
    - Merging data sections
    - Symbol resolution
    - Symbol relocation



- **Tools, libraries, debugging**
  - Cross compiler tool chain
  - Static libraries versus dynamic libraries
  - Source level debugging

Learning Objectives

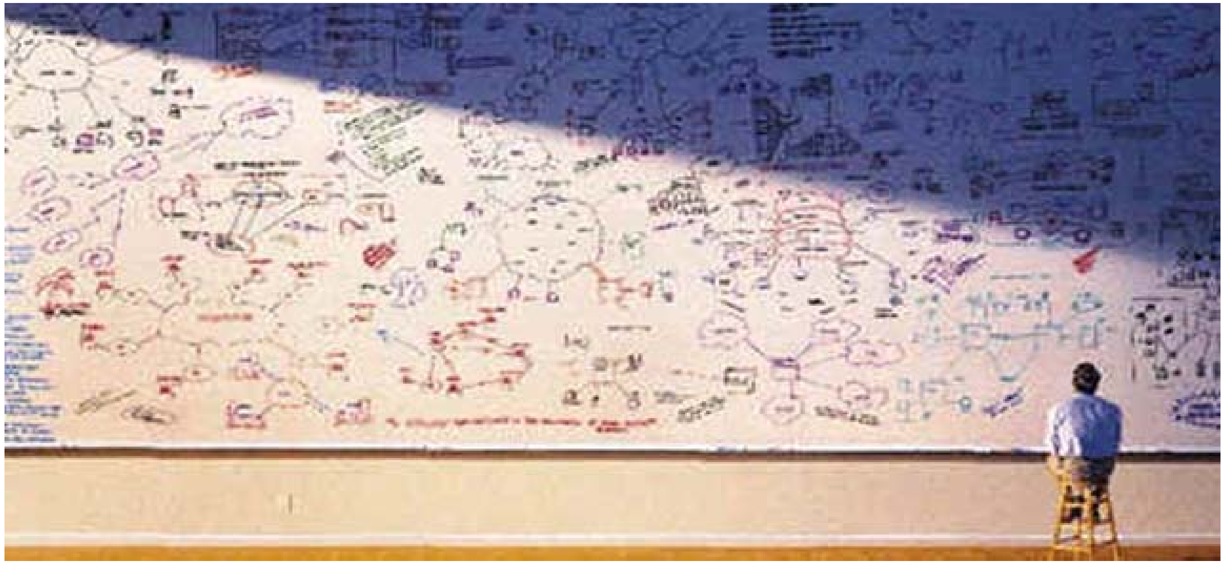
At the end of this lesson you will be able

- to explain the concepts behind modular programming
- to appropriately partition C and assembly programs into modules
- to explain the steps involved from source to the executable program
- to interpret map files of object files and executable programs
- to explain the main tasks of a linker: merging, resolution, relocation
- to explain the rules the linker applies for resolution and relocation
- to explain the difference between static and dynamic linking
- to explain the concept of source level debugging

Modular Programming – Overview

School of Engineering
InES Institute of Embedded Systems

- **Why modular programming?**
  - To manage complexity!

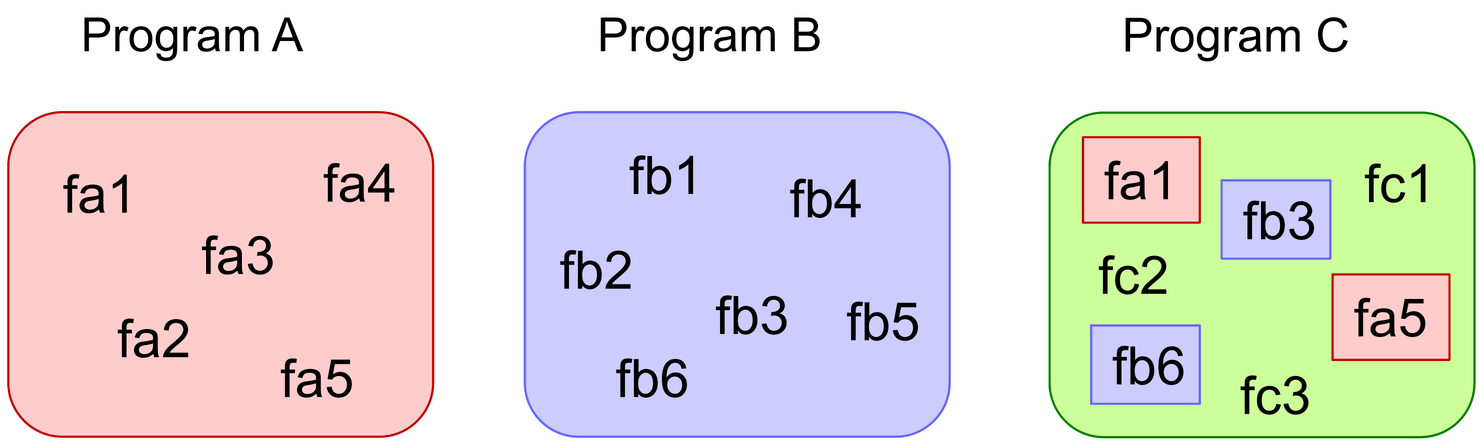


- **Basic rules**
  - Group together what belongs together
  - Split what does not belong together
  - Don’t repeat code
    - Make modules, types, and functions instead
    - Enables reusing of existing code
- **Intellectual effort**
  - There is no golden rule but established practice

1) Image source: http://www.marketingsavant.com/wp-content/uploads/2010/06/Complexity650300.jpg

Modular Programming

Example: Problem of non-modular programming



- Three single-module programs – no shared code
  - Program C needs parts from program A and B
  - Program C has copies of parts from A and B
  - Changing code and fixing bugs requires effort on multiple sources

Modular Programming

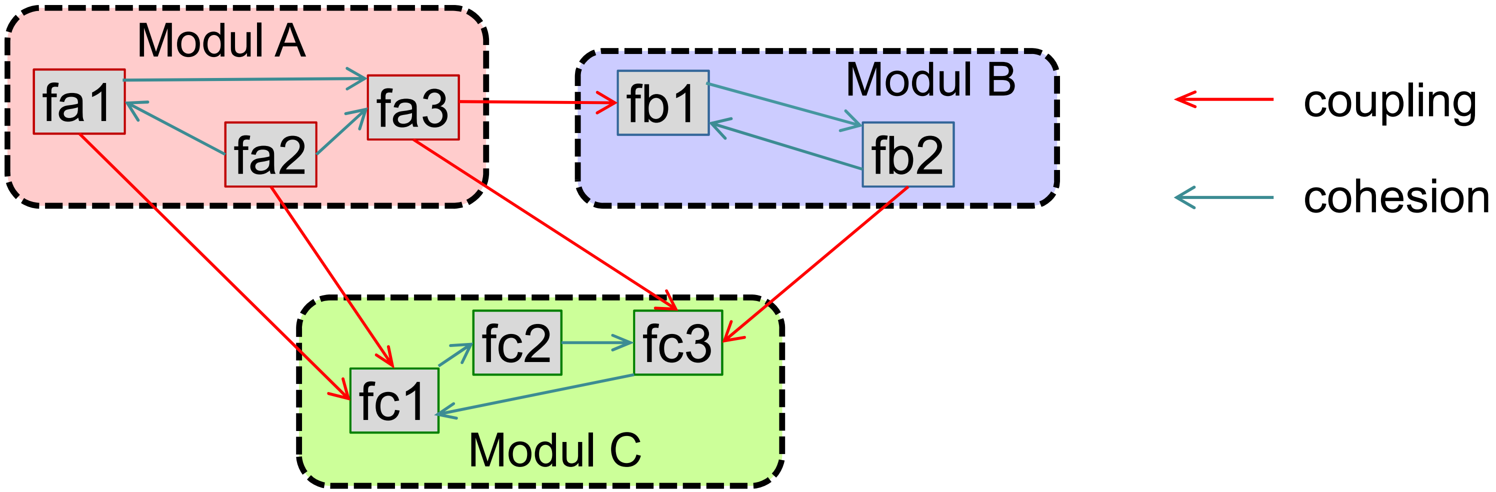
School of Engineering
InES Institute of Embedded Systems

- **Managing complexity by modular programming**

Topic	Benefits
Enable working in teams	Multiple developers working on the same source repository
Useful partitioning and structuring of the programs	Eases reusing of modules
Individual verification of each module	Benefits all users of the module
Providing libraries of types and functions	For reuse instead of reinvention
Mixing of modules that are programmed in various languages	E.g. mix C and assembly language modules
Only compile the changed modules	Speeds up compilation time

Modular Programming – Guidelines

- **High module cohesion**
  - Group together what belongs together
  - Lean external interface
  - Idea: each module fulfills a **single** defined task
- **Low module coupling**
  - Split what does not belong together
  - Little dependencies between modules



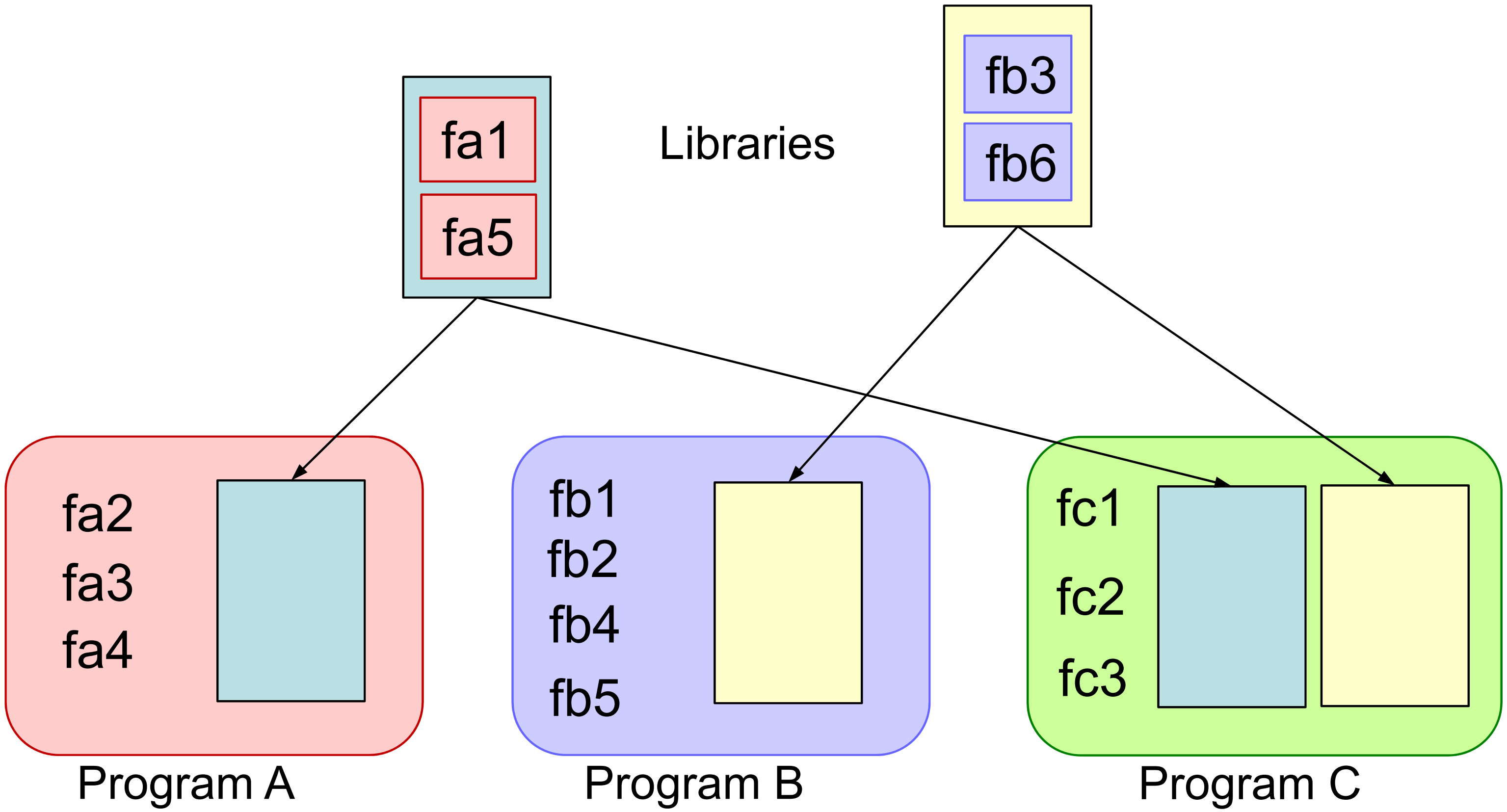
Modular Programming – Guidelines

- **Divide and conquer**
  - Partition functionality into manageable chunks
  - Hierarchical design
- **Information hiding**
  - Split interface from implementation
  - Do not disclose unnecessary details
  - Maintain freedom to change implementation details

Modular Programming – Guidelines

Reuse

- Libraries of functions and types to enable reuse



Modular Programming – Design

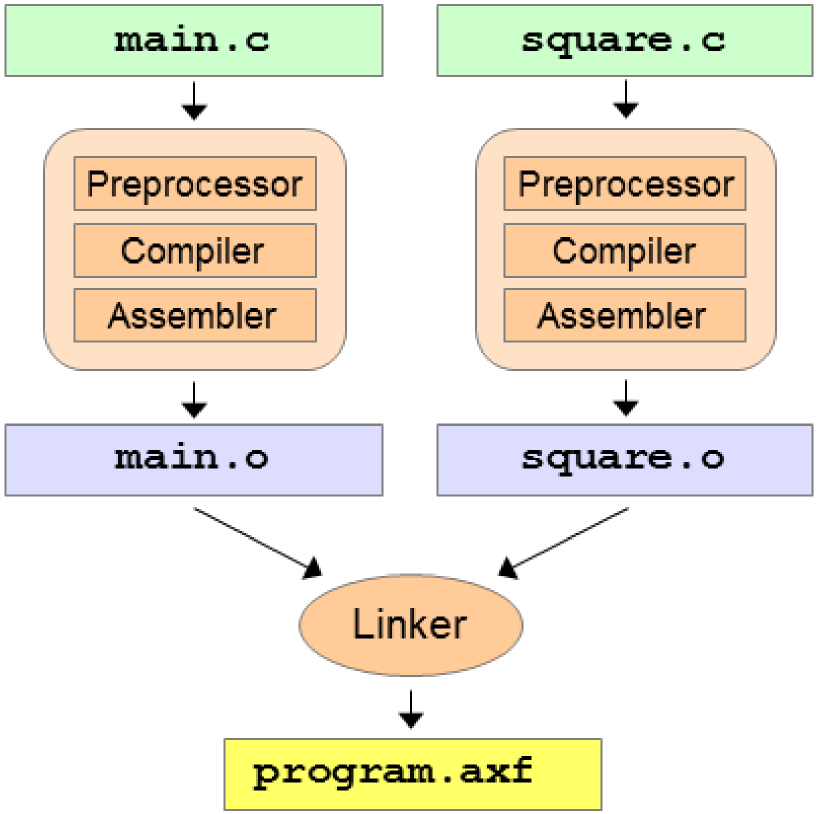
- **Module design and implementation**
  - Definition of module interface
    - Defines what functionality is available to the client of the code
  - Implementation of module
    - Provides the functionality behind the interface
    - An interface may have alternative implementations
  - Individual testing of each module
    - Modules can be tested individually if designed accordingly
  - Reuse of existing modules in new modules
    - Modules should be designed such that they can be reused

.h file

.c file

Modular Programming – Translation

- **Translation steps**
  - Compile/assemble each module
    - Results in an object file for each module (module.o[1])
  - Link all object files
    - Results in one executable file[2]



1) The file extension may vary depending on the environment
2) See later slides of this lecture

Source Code Anatomy – Overview

- **Partitioning into modules**
  - Modular programming: the whole source code base is split into multiple source files[1)]
  - Each source file defines a module
  - Each source file gets translated into one object file
  - The object files get linked into an executable file
- **Implications for source code**
  - C declarations and definitions
  - Header files to share commonly used declarations
  - Linkage of declarations and definitions
  - From C-declarations and C-definitions to assembly symbols
  - From assembly symbols to object file symbols

1) Source files are the ones that hold the core code of the module - header files are discussed later in this lecture

Source Code Anatomy

- **Challenge**
  - Modular programming requires concepts where
    - Types, functions and variables may be defined in other modules than where they are used
    - Consistency of types, functions and variables is maintained across module boundaries
- **Solution**
  - Terminology: declarations and definitions
  - Declared-before-used
  - One-definition-rule

Source Code Anatomy

C: Declaration vs. definition

- **Declaration**
  - Specifies how a name can be used[1)]

```
uint32_t square(uint32_t v); // square function defined elsewhere
extern uint32_t counter;     // counter variable defined elsewhere
struct S;                    // struct S type defined elsewhere
```

- **Definition**[2)]
  - Where a function is given with its body
  - Where memory is allocated for a variable
  - A struct type with its members

```
uint32_t square(uint32_t v) { ... } // square function definition
uint32_t counter;                   // counter variable definition
struct S { ... };                   // struct S type definition
```

1) Function and type declarations do not need the «extern» keyword

2) Each definition is implicitly also a declaration of the given name

Source Code Anatomy

- **Some C rules: what is legal and what is illegal code**
  - **Names declared before use**
    - Each name must be declared before it can be used
    - Note: a definition is also a declaration

```
uint32_t square(uint32_t v); // square is declared before use
...
out = square(in); // square is known at the point of use
```

  - **One-definition-rule**
    - A variable or function may be declared multiple times
    - But may be defined only once in the same scope[1)]

```
uint32_t in = 5; // legal first definition of the variable in
uint32_t in = 5; // illegal second definition of in
```

1) The exact rules are more elaborate

Source Code Anatomy

- **Challenge: reuse of declarations**
  - Declared-before-used may result in repeating declarations in multiple source files

```c
// program_A.c

// declaration of square
uint32_t square(uint32_t v);
...
int main(void) {
  // use of square
  res = square(a) + b;
  ...
}
```

```c
// program_B.c

// declaration of square
uint32_t square(uint32_t v);
...
int main(void) {
  // use of square
  y = square(x);
  ...
}
```

- **Maintenance issues**
  - Duplicated declarations are a consistency problem

Source Code Anatomy

Solution: use of header files

- Use a single header file instead of duplicating declarations
- Avoids copy/paste of declarations
- Maintains consistency over time

```
// square.h
#ifndef _SQUARE_H_ // incl.-
#define _SQUARE_H_ // guard

// declaration of square
uint32_t square(uint32_t v);

#endif // end of incl.-guard
```

```
// square.c
#include "square.h"

// definition of square
uint32_t square(uint32_t v)
{
  return v*v;
}
```

- Usage through **`#include`** preprocessor directive

```
// program_A.c
#include "square.h"
int main(void) {
  res = square(a) + b;
  ...
}
```

```
// program_B.c
#include "square.h"
int main(void) {
  y = square(x);
  ...
}
```

Source Code Anatomy

- **Challenge: Which names can be used by other modules?**[1)]
  - How to provide names that can be used by other modules?
  - How to inhibit use of internal names by other modules?
- **Solution: Concept of linkage in C**[1)]
  - External linkage
    - The global name is externally available for use in any modules
    - E.g. a function or a global variable
  - Internal linkage
    - The global name is only internally available for use in this module
    - E.g. a function or a global variable
  - No linkage
    - Any name that is not in the global space

1) I.e. names that are subject to symbol resolution in the linker process

Source Code Anatomy

Example: Internal and external linkage (C)

- All global names have external linkage unless defined **`static`**

```
// square.c
...
uint32_t square(uint32_t v) {
  return v*v;
}
```

square = external linkage

```
// main.c
#include "square.h"
static uint32_t a = 5;
static uint32_t b = 7;
int main(void) {
  uint32_t res;
  res = square(a) + b;
  ...
}
```

a	= internal linkage
b	= internal linkage
main	= external linkage
res	= no linkage
square	= external linkage[1)]

1) square has external linkage (no static keyword), but no definition in main.c – needs to be resolved by the linker

Source Code Anatomy

- **From C declaration/definition to assembly symbol**
  - Names given in C translate into symbols in assembly
  - C-definitions with external linkage translate into EXPORT symbols in assembly
  - C-declarations with external linkage which are used but not defined in the module translate into IMPORT symbols in assembly

```
// square.c
...
uint32_t square(uint32_t v) {
  return v*v;
}
```

→

```
; square.s
    AREA myCode,CODE,READONLY
    EXPORT square
square PROC
    MOV     r1,r0
    MULS    r0,r1,r0
    BX      lr
    ENDP
    END
```

Source Code Anatomy

School of Engineering
InES Institute of Embedded Systems

- **ARM assembly IMPORT and EXPORT keywords**
  - Linkage control
    - EXPORT declares a symbol for use by other modules
    - IMPORT declares a symbol from another module for use in this module
  - Internal symbols
    - Neither EXPORT nor IMPORT
    - Defined in this module
    - Can only be used within this module

```c
// main.c
#include "square.h"
static uint32_t a = 5;
static uint32_t b = 7;
int main(void) {
  uint32_t res;
  res = square(a) + b;
  ...
}
```

usable outside of module main

```
; main.s
      AREA myCode,CODE,READONLY
      EXPORT main
      IMPORT square
main  PROC
      LDR     r0,a_adr
      LDR     r0,[r0,#0]  ; a
      BL      square
      ...
      ENDP
a_adr DCD     a
b_adr DCD     b

      AREA myData, DATA
a     DCD     0x00000005
b     DCD     0x00000007
```

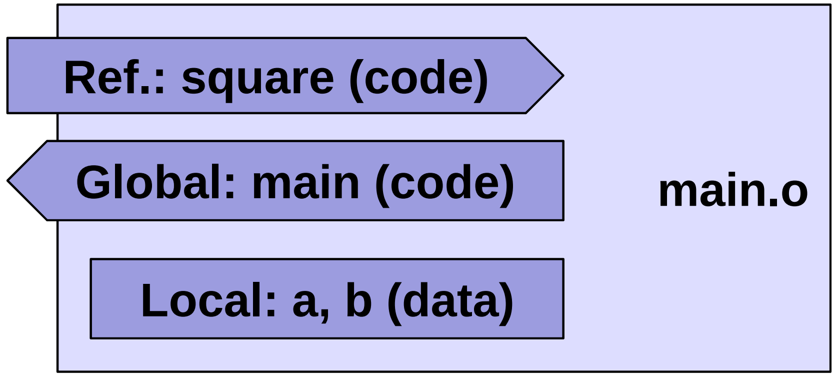
from module square

internal symbols

Source Code Anatomy

From assembly symbols to object file symbols

```
IMPORT square
...
EXPORT main
main: ...
...
a:    ...
b:    ...
```



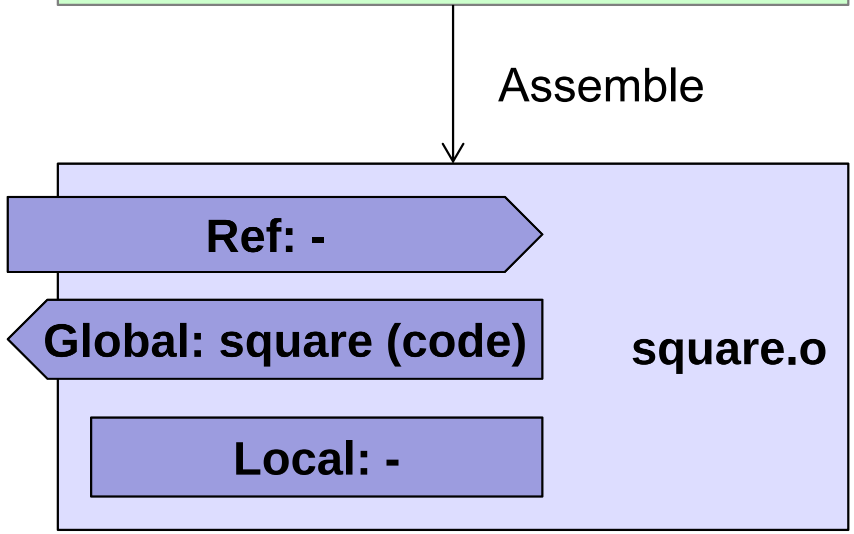
- References
  - Imported symbols from assembly code translate to global reference symbols in the object file
- Global
  - Exported symbols from assembly code translate to global symbols in the object file
- Local
  - Internal symbols from assembly code translate to local symbols in the object file

Source Code Anatomy

Example: Assembly to object file

- Exported
  - Code symbol **`square`**
- Referenced/Imported
  - None
  - No external symbol used
- Local
  - None
  - No internal symbol defined

```
; square.s
    AREA myCode,CODE,READONLY
    EXPORT square
square PROC
    MOV     r1,r0
    MULS    r0,r1,r0
    BX      lr
    ENDP
    END
```

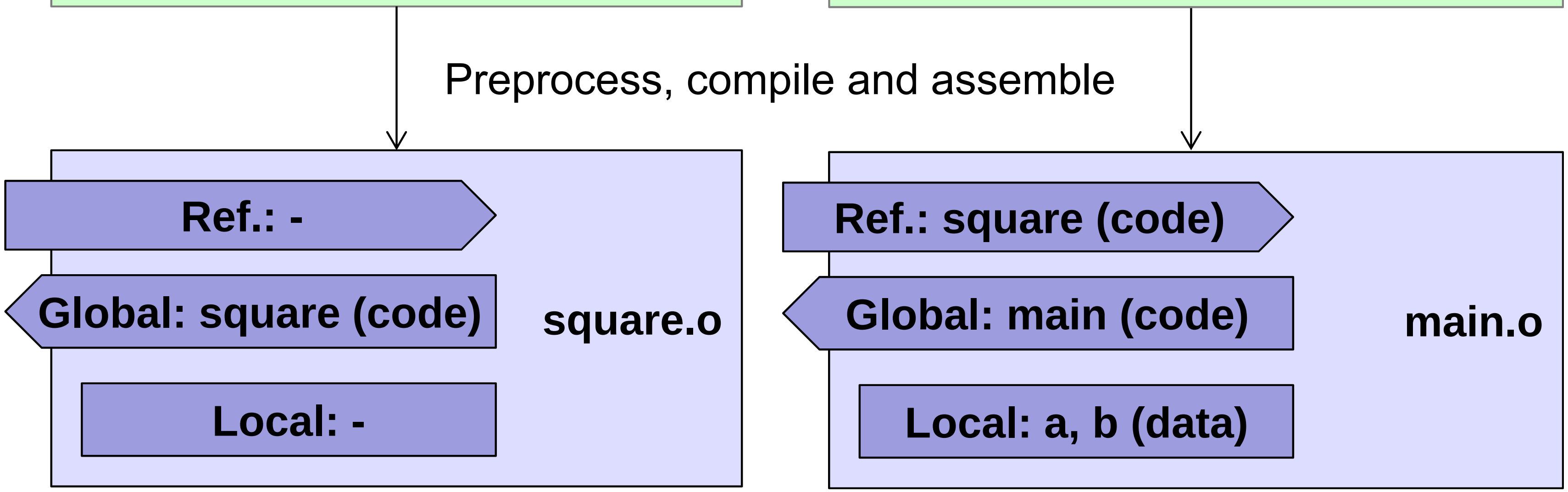


Source Code Anatomy

■ Example: C to object file

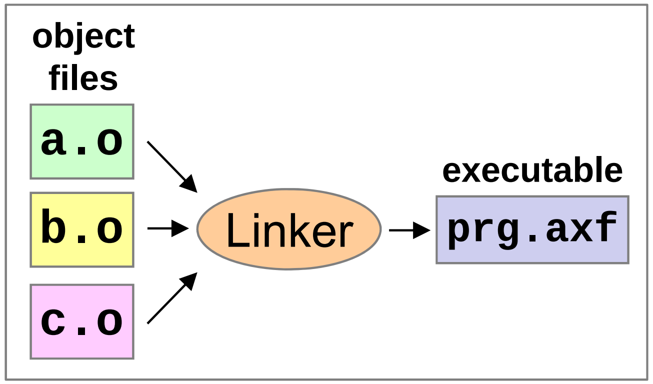
```c
// square.c
uint32_t square(uint32_t v)
{
  return v*v;
}
```

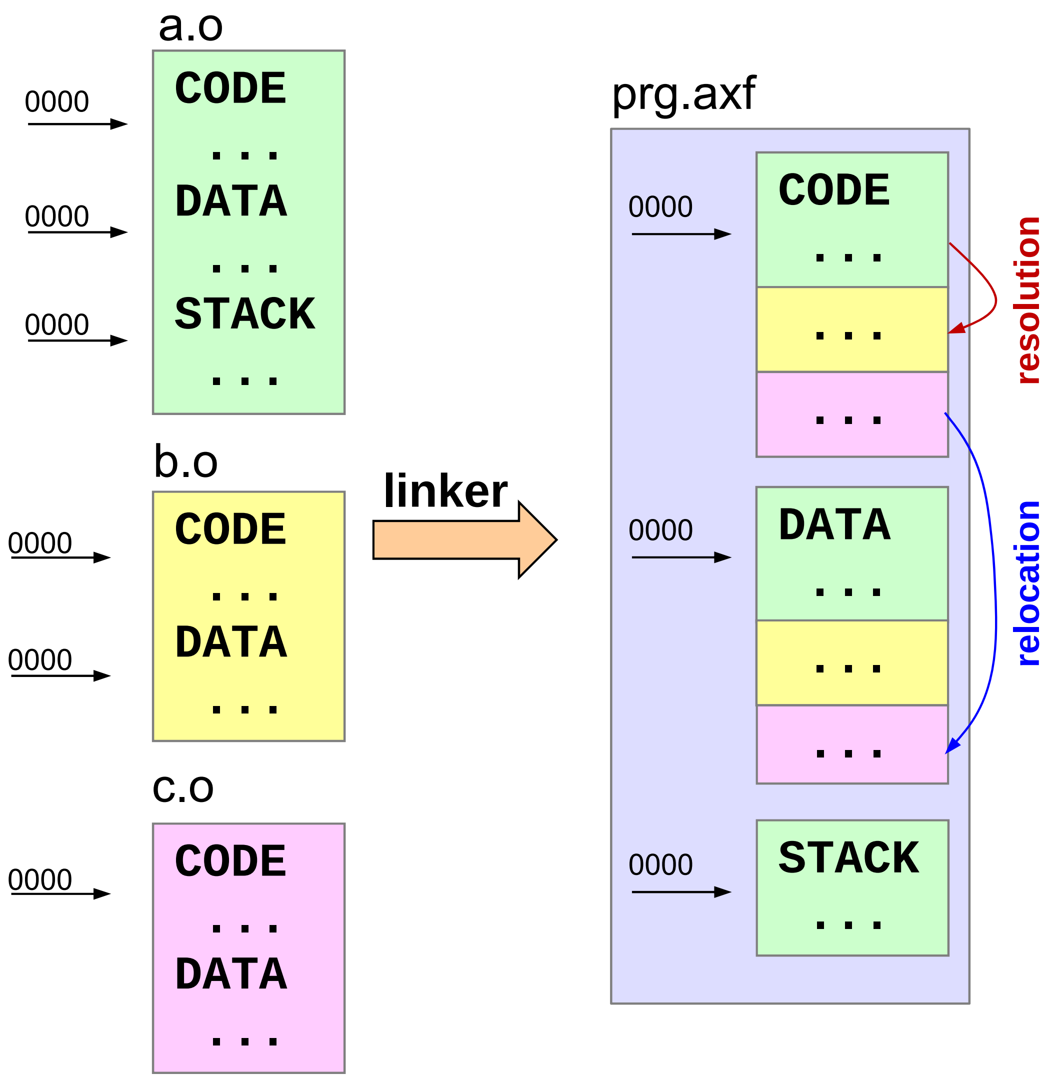
```c
// main.c
uint32_t square(uint32_t v);
static uint32_t a = 5;
static uint32_t b = 7;
int main(void) {
  uint32_t res; //stack or reg
  res = square(a) + b;
}
```



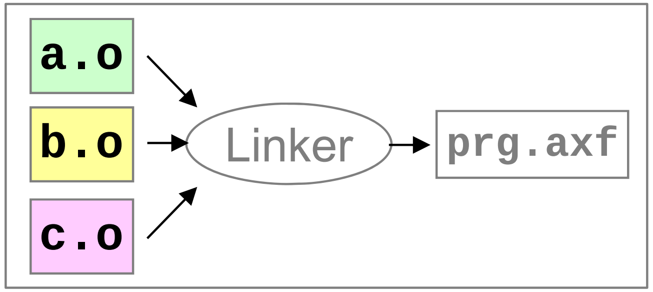
Linker – Overview

- **Linker tasks**
  - Merge code sections
  - Merge data sections
  - Symbol resolution
    - References to other modules
  - Address relocation
    - Adapt to new positions of symbols





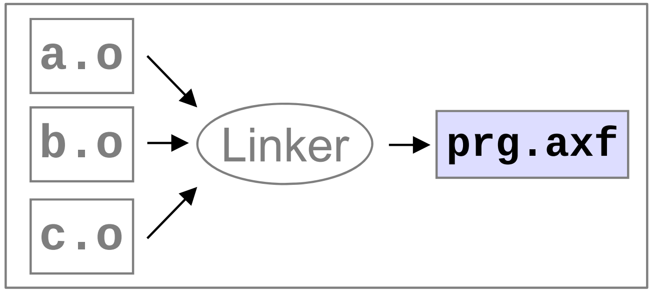
Linker Input: Object Files



- **Object files**
  - Contain all compiled data of a module
    - Code section
      - ► Code and constant data of the module, based at address 0x0
    - Data section
      - ► All global variables of the module, based at address 0x0
    - Symbol table
      - ► All symbols with their attributes like global/local, reference, etc.
    - Relocation table
      - ► Which bytes of the data and code section need to be adjusted (and how) after merging the sections in the linking process
- **ARM tool chain uses ELF for <u>object files</u>**
  - ELF = Executable and Linkable Format
    - Includes the above mentioned sections as well as further sections (e.g. string tables, debugging information, etc.)

Linker Output: Executable File

School of Engineering
InES Institute of Embedded Systems



- **Executable file**
  - Contains all linked data of the program
    - Code section
      - Code and constant data of the program
    - Data section
      - All global variables of the program
    - Symbol table
      - All symbols with their attributes like global/local, etc.
  - If the program is loaded before execution (by a loader of the hosting operating system), there might still be[1)]
    - Unresolved symbols for linking with shared (dynamic linked) libraries
    - A relocation table to move the program/data to fixed locations
- **ARM tool chain uses ELF for <u>executable file</u>**
  - File extension: AXF = **A**RM e**X**ecutable **F**ile

1) In CT1/CT2, we have no hosting OS, so we place the program sections at fixed memory location s– no loader involved

Example ELF Object File

- **square.o** [1)]
  - File section #1: code section, at base address **`0x00000000`**
  - File section #5: symbol table: **`square`** = **`global code symbol`**
  - No data section (has no global variables)
  - No relocation section (no referenced symbols in code/data)

```
File Type: ET_REL (Relocatable object) (1)
...
** Section #1 '.text' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
   Address: 0x00000000
square
        0x00000000:    4601        .F      MOV       r1,r0
        0x00000002:    4608        .F      MOV       r0,r1
        0x00000004:    4348        HC      MULS      r0,r1,r0
        0x00000006:    4770        pG      BX        lr
...
** Section #5 '.symtab' (SHT_SYMTAB)
 #  Symbol Name                Value      Bind  Sec  Type  Vis  Size
========================================================================
 6  square                     0x00000001  Gb    1  Code  Hi   0x8
```

square.o

0000 →

CODE
...

1) The output is obtained with: `c:\Keil_v5\ARM\ARMCC\bin\fromelf.exe --text -c -d -r -s -z square.o`

Example ELF Object File

School of Engineering
InES Institute of Embedded Systems

main.o (part I)

- File section #1: code section, at base address **0x00000000**
  - **`0x00000002: LDR r0, =a`** (address **a** stored at 0x14)
  - **`0x0000000a: LDR r1, =b`** (address **b** stored at 0x18)
  - **`BL square`** calls a dummy address until linked
- File section #4: data section, at base address **0x00000000**

square requires resolution

addresses of a and b require relocation

```
** Section #1 '.text' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
    Address: 0x00000000
main
        0x00000000:    b510        ..      PUSH     {r4,lr}
        0x00000002:    4804        .H      LDR      r0,[pc,#16] ; LDR r0, =a
        0x00000004:    6800        .h      LDR      r0,[r0,#0]  ; r0 = value at a
        0x00000006:    f7fffffe    ....    BL       square      ; BL needs adjustment
        0x0000000a:    4903        .I      LDR      r1,[pc,#12] ; LDR r1, =b
        0x0000000c:    6809        .h      LDR      r1,[r1,#0]  ; r1 = value at b
        0x0000000e:    1844        D.      ADDS     r4,r0,r1
        0x00000010:    2000        .       MOVS     r0,#0
        0x00000012:    bd10        ..      POP      {r4,pc}

        0x00000014:    00000000    ....    DCD    0 ; address a will be stored here
        0x00000018:    00000000    ....    DCD    0 ; address b will be stored here
** Section #4 '.data' (SHT_PROGBITS) [SHF_ALLOC + SHF_WRITE]
     Address: 0x00000000
        0x00000000:    00000005                     ; value at a = 5
        0x00000004:    00000007                     ; value at b = 7
```

main.o

0000	CODE
	...
0000	DATA
	...

Example ELF Object File

main.o (part II)

- File section #6: symbols:
  - **a**: local data section symbol, at offset 0x00000000
  - **b**: local data section symbol, at offset 0x00000004
  - **`main`**: global code section symbol, at offset 0x00000000 (LSB set: Thumb code)
  - **`square`**: global code section symbol, referenced (no definition in main.o)

```
...
** Section #6 '.symtab' (SHT_SYMTAB)
 #  Symbol Name                   Value       Bind  Sec  Type  Vis  Size
========================================================================
 7  a                             0x00000000  Lc      4  Data  De   0x4
 8  b                             0x00000004  Lc      4  Data  De   0x4
11  main                          0x00000001  Gb      1  Code  Hi   0x14
12  square                        0x00000000  Gb    Ref  Code  Hi
...
```

Example ELF Object File

School of Engineering
InES Institute of Embedded Systems

main.o (part III)

- File section #7: relocation table:
  - Relocation at code address **0x00000006**:
    - ► Modify the BL call to branch to the symbol **square**
  - Relocation at code address **0x00000014**:
    - ► Set the absolute 32 bit value of the symbol **a**
  - Relocation at code address **0x00000018**:
    - ► Set the absolute 32 bit value of the symbol **b**

Relocation table section

```
...
** Section #7 '.rel.text' (SHT_REL)
 #      Offset     Relocation Type               Wrt Symbol
========================================================================
 0  0x00000006   10 R_ARM_THM_CALL            12 square
 1  0x00000014    2 R_ARM_ABS32                7 a
 2  0x00000018    2 R_ARM_ABS32                8 b
...
```

Affected code section locations

```
** Section #1 '.text' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
...
0x00000006:    f7fffffe    ....    BL       square      ; BL needs adjustment
...
0x00000014:    00000000    ....    DCD      0 ; address a will be stored here
0x00000018:    00000000    ....    DCD      0 ; address b will be stored here
```

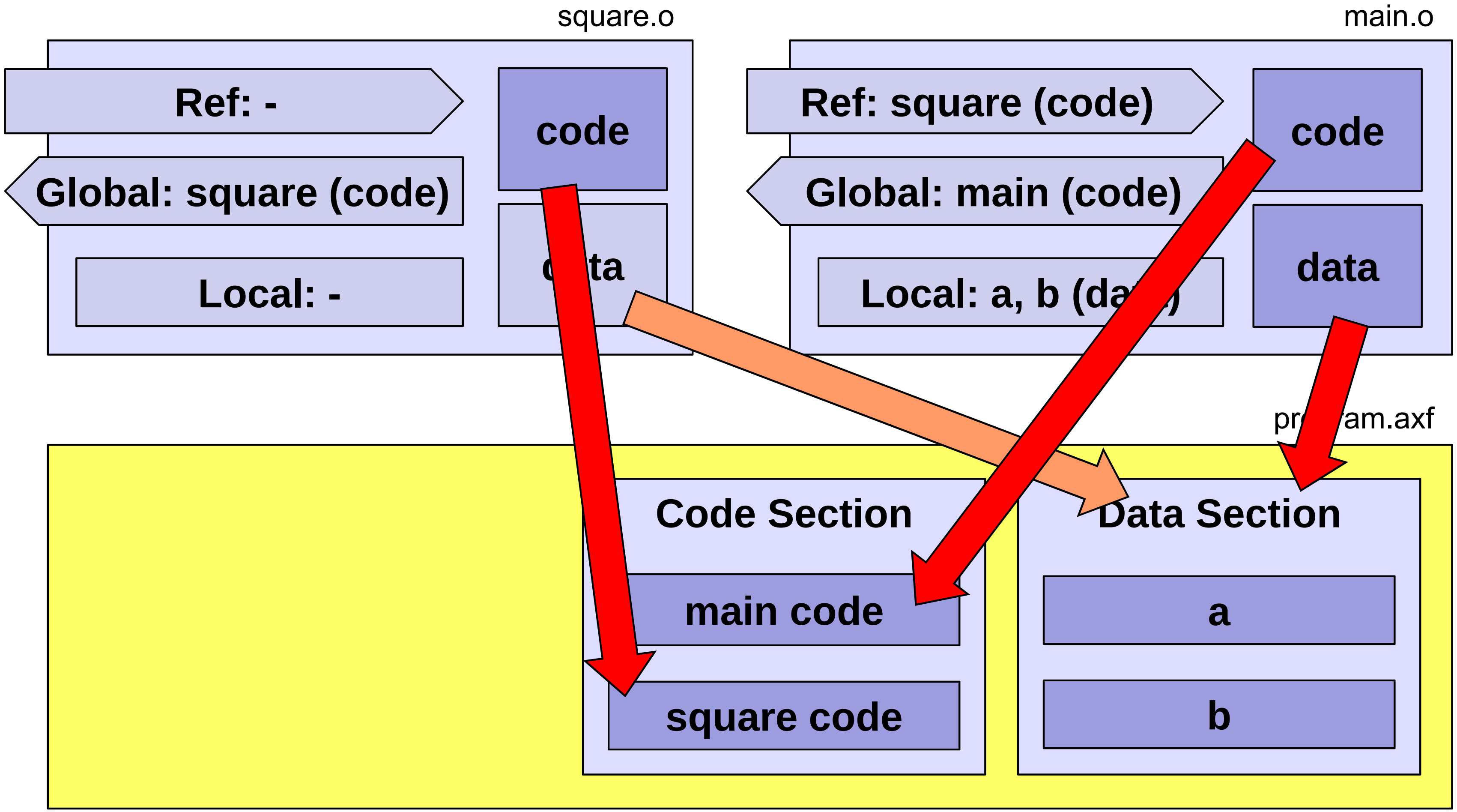
Tasks of a Linker – Overview

- **Tasks of a Linker**
  - **Merge object file data sections**
    - Place all data sections of the individual object files into one data section of the executable file
  - **Merge object file code sections**
    - Place all code sections of the individual object files into one code section of the executable file
  - **Resolve used external symbols**
    - Search missing addresses of used external symbols
  - **Relocate addresses**
    - Adjust used addresses since merging the sections invalidated the original addresses[1)]

1) The linker places the sections depending on the target system, the given command line arguments and the scatter file (if given). A scatter file tells at which absolute addresses the code and data sections are placed.
In an OS-hosted environment, addresses for code and data sections are at a fixed “virtual” address. The loader then places the sections into suitable virtual process memory.

Tasks of a Linker

- **Merge data sections and code sections of the modules**
  - Place one section after the other
    - Note: in this example, square.o has no data section



Tasks of a Linker – Example

Example: Merge code sections

- Merging code sections of main.o and square.o
  - Offset for first code section is **0x00000000** (main.o)
  - Offset for next code section is **0x0000001C** (square.o)

```
0x00000000 B510       PUSH     {r4,lr}                 code (main.o)
0x00000002 4804       LDR      r0,[pc,#16]
0x00000004 6800       LDR      r0,[r0,#0x00]
0x00000006 f7fffffe   BL       square
0x0000000A 4903       LDR      r1,[pc,#12]
0x0000000C 6809       LDR      r1,[r1,#0x00]
0x0000000E 1844       ADDS     r4,r0,r1
0x00000010 2000       MOVS     r0,#0x00
0x00000012 BD10       POP      {r4,pc}
0x00000014 00000000   DCD      0x00000000
0x00000018 00000000   DCD      0x00000000
```

```
0x0000001C 4601       MOV      r1,r0                   code (square.o)
0x0000001E 4608       MOV      r0,r1
0x00000020 4348       MULS     r0,r1,r0
0x00000022 4770       BX       lr
```

- No resolution nor relocation done yet

Tasks of a Linker – Example

Example: Merge data sections

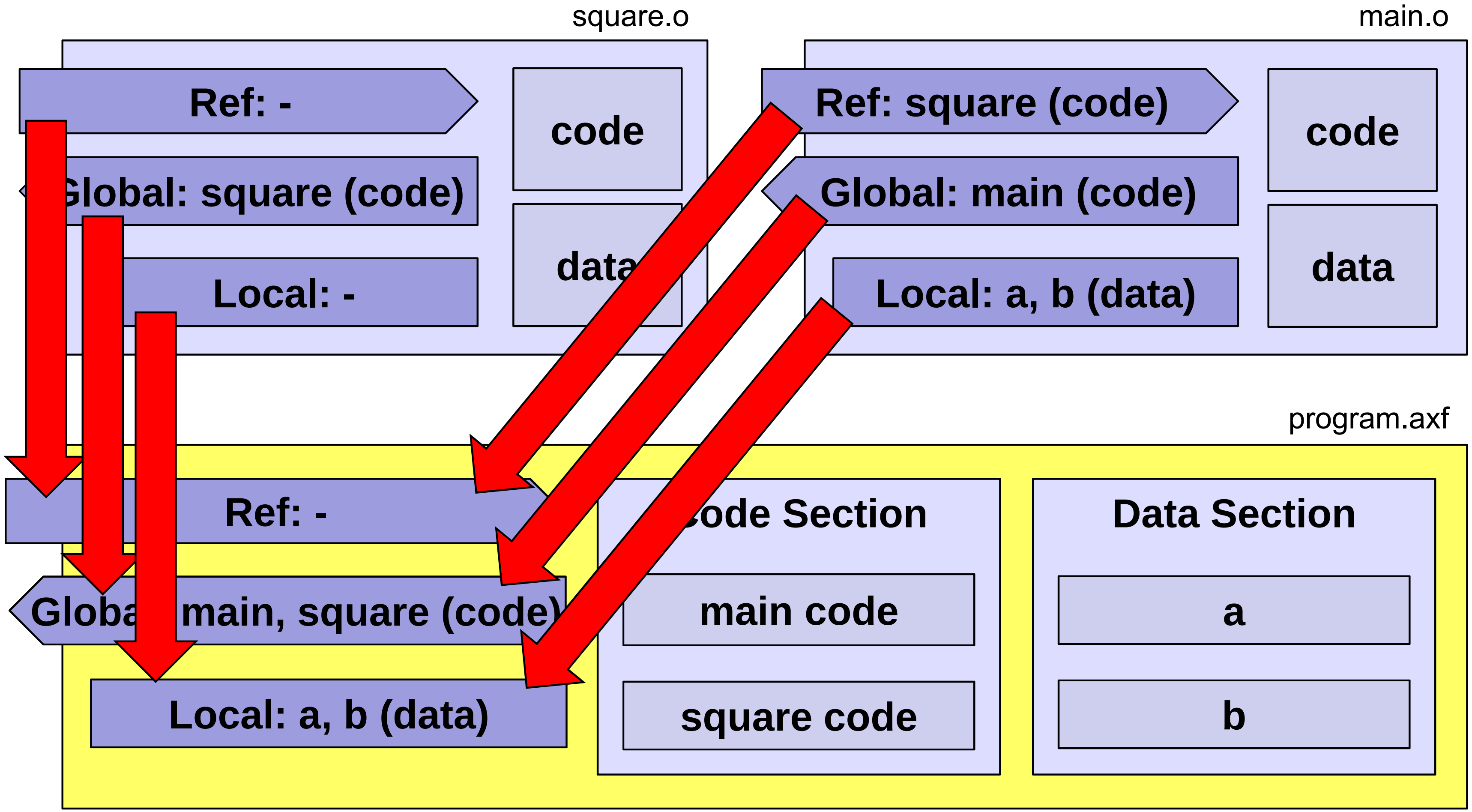
- Merging data sections of main.o and square.o
  - Offset for first data section is **0x00000000** (main.o)
  - There is no further data section (square.o has no global data)

```
0x00000000 05000000   DCD   5                                   data (main.o)
0x00000004 07000000   DCD   7
```

- No resolution nor relocation done yet

Tasks of a Linker – Resolution

- **Resolve referenced symbols**
  - Merge the symbol tables
  - Resolve references within symbol table



Tasks of a Linker – Example

Example: Resolve symbols

- Merging symbol table sections of main.o and square.o

symbols (main.o)

#	Symbol Name	Value	Bind	Sec	Type	Vis	Size
7	a	0x00000000	Lc	4	Data	De	0x4
8	b	0x00000004	Lc	4	Data	De	0x4
11	main	0x00000001	Gb	1	Code	Hi	0x14
12	square	0x00000000	Gb	Ref	Code	Hi	

symbols (square.o)

#	Symbol Name	Value	Bind	Sec	Type	Vis	Size
6	square	0x00000001	Gb	1	Code	Hi	0x8

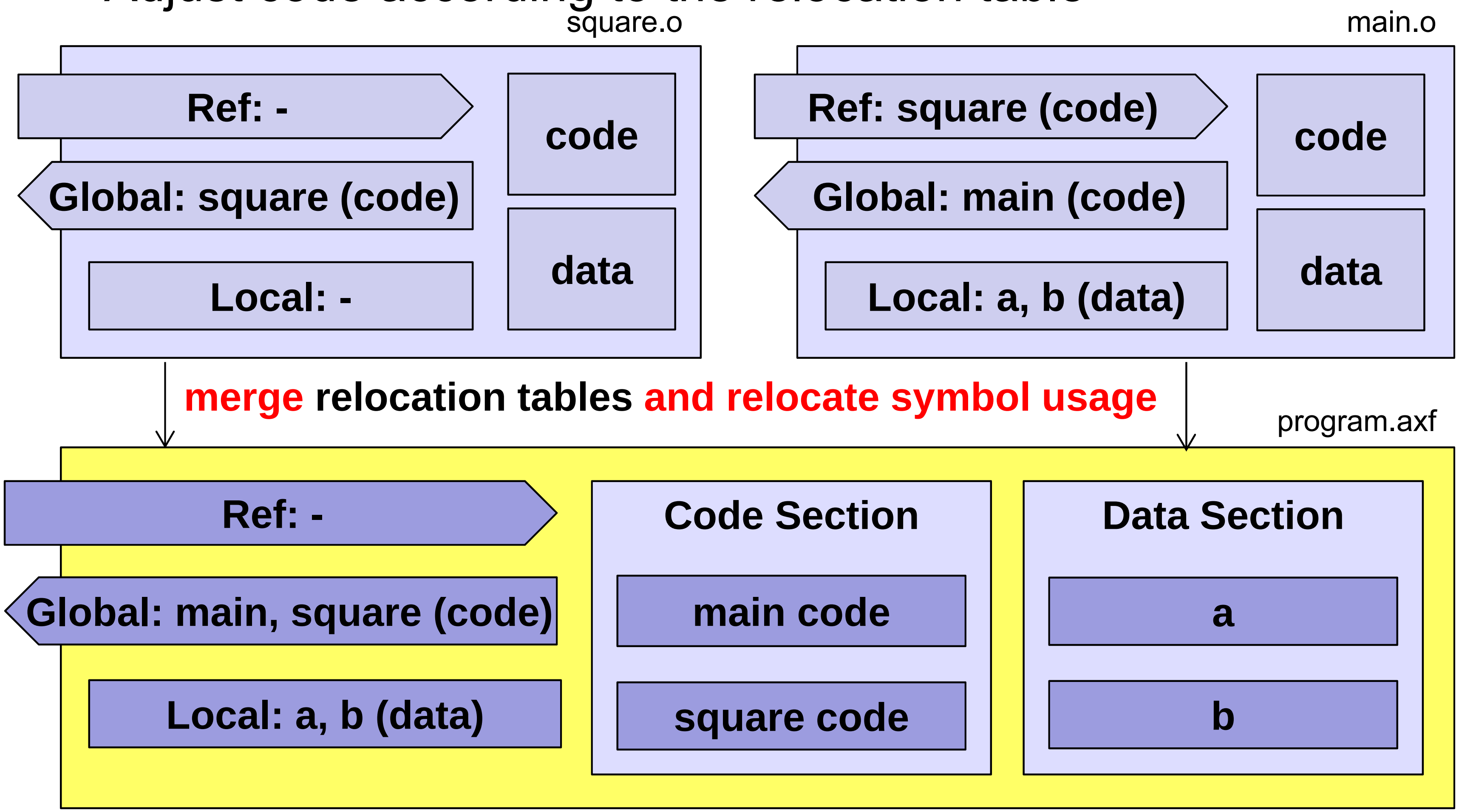
resolved symbols

#	Symbol Name	Value	Bind	Sec	Type	Vis	Size	
20	a	0x00000000	Lc	4	Data	De	0x4	(main.o)
21	b	0x00000004	Lc	4	Data	De	0x4	(main.o)
186	main	0x00000001	Gb	1	Code	Hi	0x14	(main.o)
187	square	0x00000000	Gb	1	Code	Hi	0x8	(square.o)

- The relative values of the symbols within the modules are not yet relocated to global addresses. Therefore, the linker needs to remember for which module/section the relative address is given
- No relocation done yet

Tasks of a Linker – Relocation

- **Relocate usage of symbols**
  - Merge the relocation tables
  - Relocate code and data section, symbols, relocation table
  - Adjust code according to the relocation table



Tasks of a Linker – Example

Example: Relocate sections and usage of symbols (I)

- Merging relocation table sections of main.o and square.o
  - square.o has no relocation table

merged relocation

```
 #      Offset     Relocation Type                  Wrt Symbol
========================================================================
 0   0x00000006    10 R_ARM_THM_CALL             12 square                  (main.o)
 1   0x00000014     2 R_ARM_ABS32                 7 a                       (main.o)
 2   0x00000018     2 R_ARM_ABS32                 8 b                       (main.o)
```

- The relative address of the relocation table within the modules is not yet adjusted to global addresses, therefore, needs to remember for which module and which section the relative address is given

Tasks of a Linker – Example

- **Example: Relocate sections and usage of symbols (II)**

  1) Relocate
     - Sections
     - Symbols
     - Relocation offsets

     Relocation calculations
     - new value = global base + merge offset + module relative offset
     - E.g. symbol **b**:
       - ► global base = internal SRAM = 0x20000000
       - ► merge offset = 1st in merged data section = 0x00000000
       - ► module relative offset = b is the 2nd variable after a = 0x00000004
       - ► new value for symbol b = <u>0x20000004</u>
     - E.g. symbol **`square`** if user code (like main) starts at 0x08000254
       - ► 0x08000254 + 0x0000001C + 0x00000000 = <u>0x08000270</u>

  2) Adjust the code according to the relocation table

Tasks of a Linker – Example

Example: Relocate sections and usage of symbols (III)

- **Relocated** code sections

```
0x08000254 B510       PUSH     {r4,lr}              ; main
0x08000256 4804       LDR      r0,[pc,#16]
...
0x08000270 4601       MOV      r1,r0                ; square
0x08000272 4608       MOV      r0,r1
...
```

- **Relocated** data sections

```
0x20000000 00000005   DCD   5                       ; value of a
0x20000004 00000007   DCD   7                       ; value of b
```

- **Relocated** symbols

```
 20  a                          0x20000000   Lc    4  Data  De   0x4
 21  b                          0x20000004   Lc    4  Data  De   0x4
186  main                       0x08000255   Gb    1  Code  Hi   0x14
187  square                     0x08000270   Gb    1  Code  Hi   0x8
```

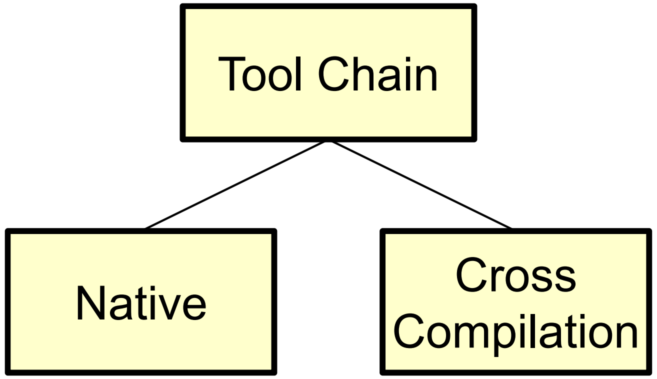
- **Relocated** relocation table entries

```
 0   0x0800025A    10 R_ARM_THM_CALL            12 square
 1   0x08000268     2 R_ARM_ABS32                7 a
 2   0x0800026C     2 R_ARM_ABS32                8 b
```

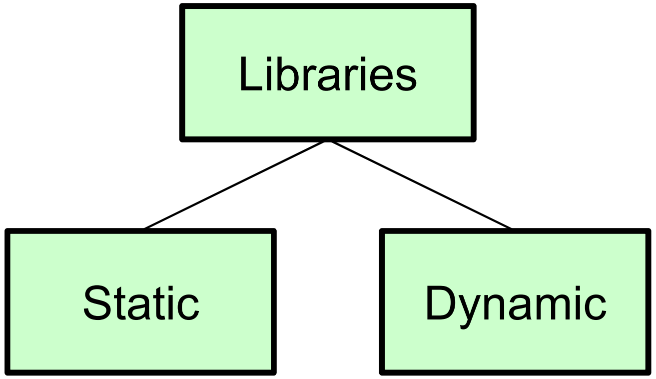
- **Adjusted** code locations according to relocation table

```
...
0x0800025A F000F809   BL.W     square      ; 0x08000270
...
0x08000268 20000000   DCD      0x20000000
0x0800026C 20000004   DCD      0x20000004
```

Tool Chain, Libraries, Debugging



- **Tool chain**
  - Native tool chain
  - Cross compiler tool chain
- **Libraries**
  - Libraries
    - Collection of object files
  - Static libraries
    - Linked into the executable at link time
  - Dynamic or shared libraries
    - Executable is linked at loading time with the shared library



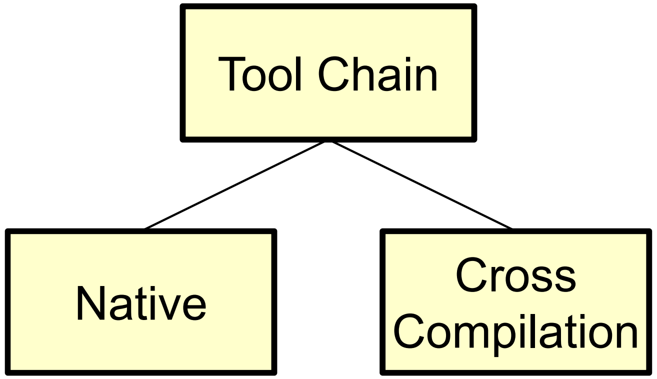
- **Debugging**
  - Single step and breakpoints
  - Source level debugger

```
REG_TIMx_PSC(TIM4_BASE)   = (F_84MHZ / F_10KHZ)-1;
REG_TIMx_ARR(TIM4_BASE)   = F_10KHZ - 1;
REG_TIMx_CR1(TIM4_BASE)   = 0x0;
REG_TIMx_CR1(TIM4_BASE)  |= DOWNCOUNT;
REG_TIMx_DIER(TIM4_BASE) |= UIE;
REG_TIMx_CR1(TIM4_BASE)  |= CEN;
```

Tool Chain

Tool chain

- Minimal view
  - The set of tools that is required to create from source code an executable for a given environment



- Native tool chain
  - Builds for the same architecture where it runs on
- Cross compiler tool chain
  - Builds for another architecture than the one it runs on
  - E.g. build in KEIL (on Windows) for the CT Board (bare-metal ARM)

- Professional view: there is more than the “compiler & linker”:
  - Editing tools (IDE), revision control tools, documentation tools, testing tools, build tools, deployment tools, issue tracking tools, …

Cross Compiler Tool Chain

- **KEIL IDE – Integrated Development Environment**
  - UI Frontend for editing, compiling and debugging

- **Cross compiler tool chain**
  - Produces executable programs for a target system which is different from the host system of the tool chain (e.g. compile on a Windows PC for an ARM platform).
  - Behind the scene, KEIL IDE employs a cross compilation tool chain
    - **`armcc`** ARM C/C++ Compiler (including Preprocessor)
    - **`armasm`** ARM Assembler Compiler
    - **`armar`** ARM Library manager
    - **`armlink`** ARM Linker
    - **`fromelf`** ARM Image conversion and dumper tool

Libraries

- **Libraries in general**
  - Collection of object files
  - May speedup linking
    - May provide an overall prepared (sorted) symbol table
  - Linking with libraries may result in smaller code
    - Libraries may provide only the really needed parts of the sections
    - Linking with plain object files always links all and the whole sections
  - Created by a librarian tool (e.g. armar for our environment)
  - May replace one library by another one
    - E.g. at evaluation time have a working model, at production time have a high-performance library of the same funcitionality

Libraries

- **Static libraries**
  - Executable is completely linked with a static library at link time
  - The resulting executable is self contained
    - No need for any other libraries at run time
  - Benefit
    - Self contained
    - No version issues in the run environment
    - No support needed from any hosting OS
  - Drawback
    - Larger executables compared to dynamically linked libraries
    - No possibility to share common code between different executables
    - Cannot replace broken shared code with a new version of the library
- **KEIL/ARM**
  - Static libraries are used in the ARM cross compilation environment

Libraries

- **Dynamic or shared libraries**
  - Executable is not linked with a dynamic library at link time
  - The resulting executable is not self contained
    - Needs other libraries at run time
    - Loader of hosting OS links at load time with the shared libraries
  - Benefit
    - Smaller executables compared with static libraries
    - Can replace shared libraries
  - Drawback
    - May result in versioning problems at load time
      - Well known "DLL-Hell" from MS Windows environment
- **Windows/Unix/OSx**
  - With PC OS support you have generally both libraries
    - Static: libX.a, dynamic: libX.so (Unix/Linux/OSx), libX.dll (Windows)

Debugging

- **Single stepping**
  - Support by the HW (stops processor, provides register access)
  - Support by SW (swap instructions with a breakpoint instruction)
- **Source level debugging**
  - Source level debugging needs mapping between
    - Machine address and source code line
    - Memory locations and source code types
  - Mapping information is often also provided in object files (e.g. in the ELF files)
    - Also depends on all linking steps (merging section, resolve symbols, relocate symbol usage)
    - On Windows, this information is provided in a separate file (PDB)

Conclusions

School of Engineering
InES Institute of Embedded Systems

- **Modular programming**
  - Crucial concept of software development
  - There is no golden rule but established practice
  - C supports this by use of header files and compilation into object files
- **Linker**
  - Combines object files into executable by merging sections, resolve referenced symbols and relocating all symbols and code
  - Is ignorant of the used programming language
- **Tools**
  - Tool chains and further tools build the working environment
- **Debugging**
  - Source-level debugging is a crucial tool to analyze and fix bugs