

# Embedded Software Development Tools



Simon, Ch9

Compilers Linker/Locators, address resolution, MAP files, getting software into the target system.

# Chain of Events

---

- When you produce an executable, several things happen:
  1. Compile
    - Translates source code (C) into object code
    - Also, preprocessing, optimizing, semantic analysis
    - Each source file creates one object file
  2. Link
    - Combines one or more object files into an executable
  3. Run

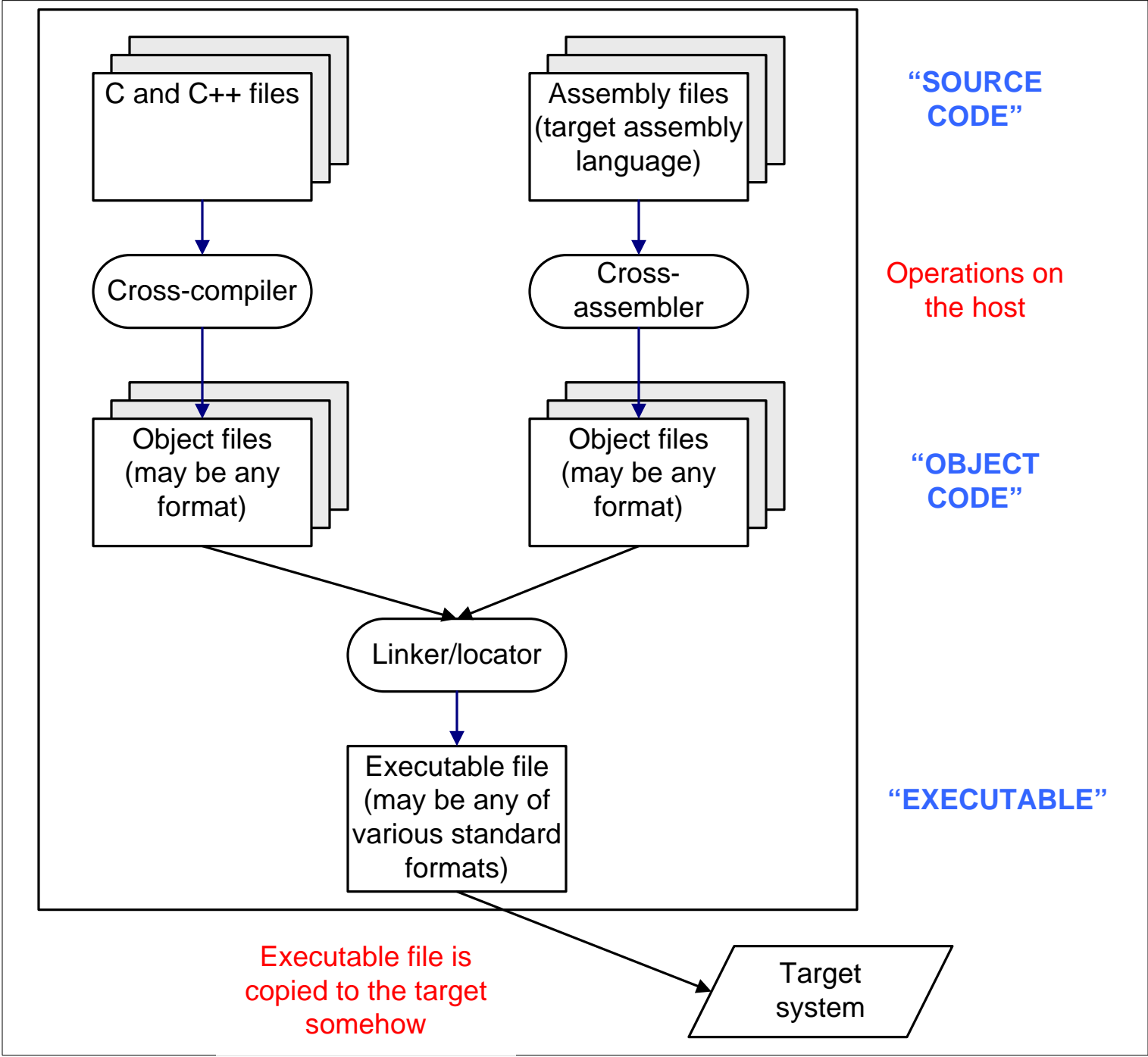
# Host and Target Machines

- Host: computer where tools are run
- Target: processor you are shipping
- Native tools – run on the host
  - E.g. Windows NT compiler produces binary Pentium instructions (great for Pentium, worthless for Motorola 68000)
- Cross-Compilers
  - Runs on your host, but produces binary instructions for your target from your C/C++ programs.
- Cross-Assemblers
  - Runs on your host, but produces binary instructions for your target from your assembly programs.
- Tool Chains
  - Named because output of one tool becomes the input for the next tool
  - Tools compatible with a particular target are called a “Tool Chain” and are available from different vendors.

# Why all these Tools?

---

- Why Won't Any Compiler Work?
  - Theoretically, it should.
  - Problems occur with declarations (older styles, using functions without declaring them, undefined behavior in the standards, etc.)
  
- Why Won't Host Code Work on Target?
  - Different size ints
  - Different structure packing
  - Ability to access odd/even addresses
  - Different peripherals and hardware



# Linker/locators: Address Resolution

---

- Many microprocessor instructions contain addresses of operands (variables, function calls)
- Hooking up all the addresses is called Address Resolution
- When each file compiled, addresses are not known, so flags left as placeholders.
- The linker combines all files, so it can replace placeholders with actual address values.
- Loader/locator also must figure out starting memory address for executable

```
ABBOTT.C
int idunno;
...
whosonfirst(idunno);
...
```

```
COSTELLO.C
...
int whosonfirst (int x)
{
...
}
```

"SOURCE  
CODE"

Compiler

Compiler

```
ABBOTT.OBJ
...
MOVE R1, (idunno)
CALL whosonfirst
...
```

```
COSTELLO.OBJ
...
whosonfirst:
...
```

"OBJECT  
CODE"

Linker

```
HAHAHA.EXE
...
MOVE R1, 2388
CALL 1547
...
1547 MOVE R1, R5
...
2388 (value of idunno)
```

Loader/  
Locator

```
Memory
HAHAHA.EXE
...
MOVE R1, 22388
CALL 21547
...
21547 MOVE R1, R5
...
22388 (value of idunno)
```

# Locating Program Components

---

- Programs contain ROM and ROM elements - how does the tool chain know where to put each?
- Programs are divided into **segments**.
- Segment: individual piece of code that the locator can place separately in memory.
- Startup code: usually asm at magic start location in memory
- Each module usually divided into several segments:
  - The instructions: "code"
  - Uninitialized data: "udata" (e.g., int barney;)
  - Constant strings: "strings" (e.g., "Press start")
  - Initialized data: "idata" (e.g., char fred = 0x55;)

# Intel Hex File Format

```

. . .
:101060000FF908193E03400FAA9077B021227BE901B
:10107000093E034A9077B02122908193E908193E908
:101070000EF908400FAA9077B021227BE901B3E9081
:05100900027BE1212C48E
:011095002238
:011096002237
:10109700400FAA9400FAA9400FAA993E034A9077B02
:0410A7001281C622CA
:0110AB002222
:1010AC00EF908400FAA9077B021227BE901B3E9081
. . .
. . .

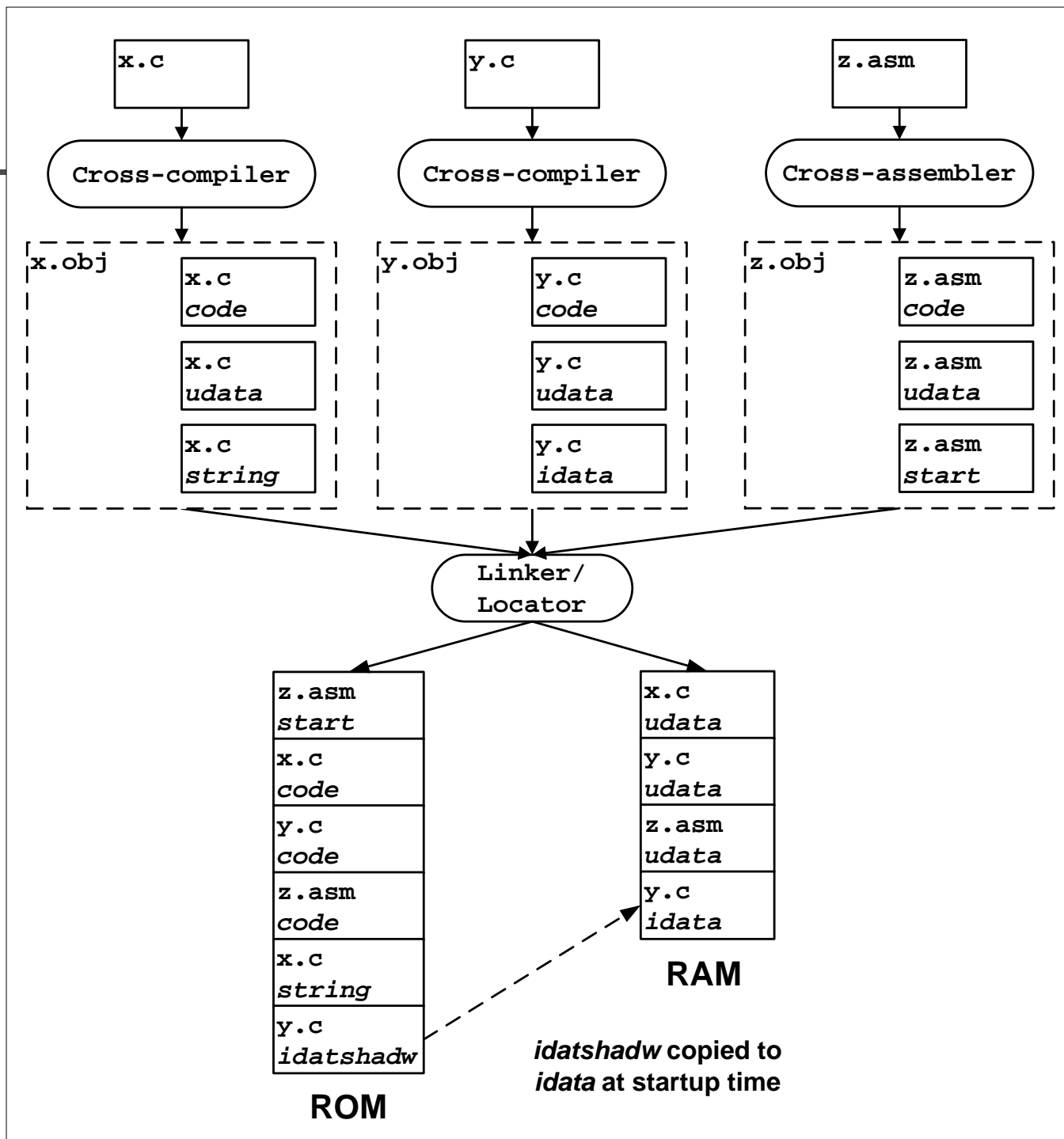
```



# Sample Compile-Link-Locate

---

- Program made up of 3 files: x.c, y.c, z.asm
- x.c contains
  - Instructions code
  - Uninitialized data udata
  - Constant strings strings
- y.c contains
  - Instructions code
  - Uninitialized data udata
  - Initialized data idata
- z.asm contains
  - Asm functions code
  - Start-up code startup
  - Uninitialized data udata



# Placing Segments in Memory

Instructions to the locator:

```
-CZSTART, IVECS, CODE=0
-ZIDATA, UDATA, CSTACK=8000
```

Resulting program

0000	CSTART
	IVECS
	CODE
	(unused)
8000	IDATA
	UDATA
	CSTACK
	(unused)

# Our Memory Map - Flashing LED

## memory organization

		MSP430F200x	MSP430F201x
Memory Main: Interrupt vector Main: code memory	Size	1KB Flash	2KB Flash
	Flash	0FFFFh-0FFC0h	0FFFFh-0FFC0h
Information memory	Flash	0FFFFh-0FC00h	0FFFFh-0F800h
	Size	256 Byte	256 Byte
	Flash	010FFh - 01000h	010FFh - 01000h
RAM	Size	128 Byte	128 Byte
		027Fh - 0200h	027Fh - 0200h
Peripherals	16-bit	01FFh - 0100h	01FFh - 0100h
	8-bit	0FFh - 010h	0FFh - 010h
	8-bit SFR	0Fh - 00h	0Fh - 00h

SEGMENT	SPACE	START ADDRESS	END ADDRESS	SIZE	TYPE	ALIGN
=====	=====	=====	=====	=====	=====	=====
DATA16_AN		0021	- 0022	2	rel	0
		0120	- 0121	2		
CSTACK		024E	- 027F	32	rel	1
CSTART		F800	- F811	12	rel	1
CODE		F812	- F837	26	rel	1
RESET		FFFE	- FFFF	2	rel	1

# Locator Maps

- Shows all locations in memory - check to make sure the locator matches your target!
- Maps are great for debugging - memory overwrites, etc.
- IAR->Project->Options->
  - C/C++->List->Output List File
  - Linker->List->Generate Linker Listing
- Look at output files <project>.map

# Getting Software Into Target

---

- PROM programmers - actual ROM
- ROM emulators - hardware thinks it's a ROM
- In-circuit emulators - replaces entire processor on hardware board
- FLASH - if target stores program separately
- Monitors - program in target to receive program  
- for debugging mainly