

Herramientas de desarrollo de software embebido

Sistemas embebidos para tiempo real

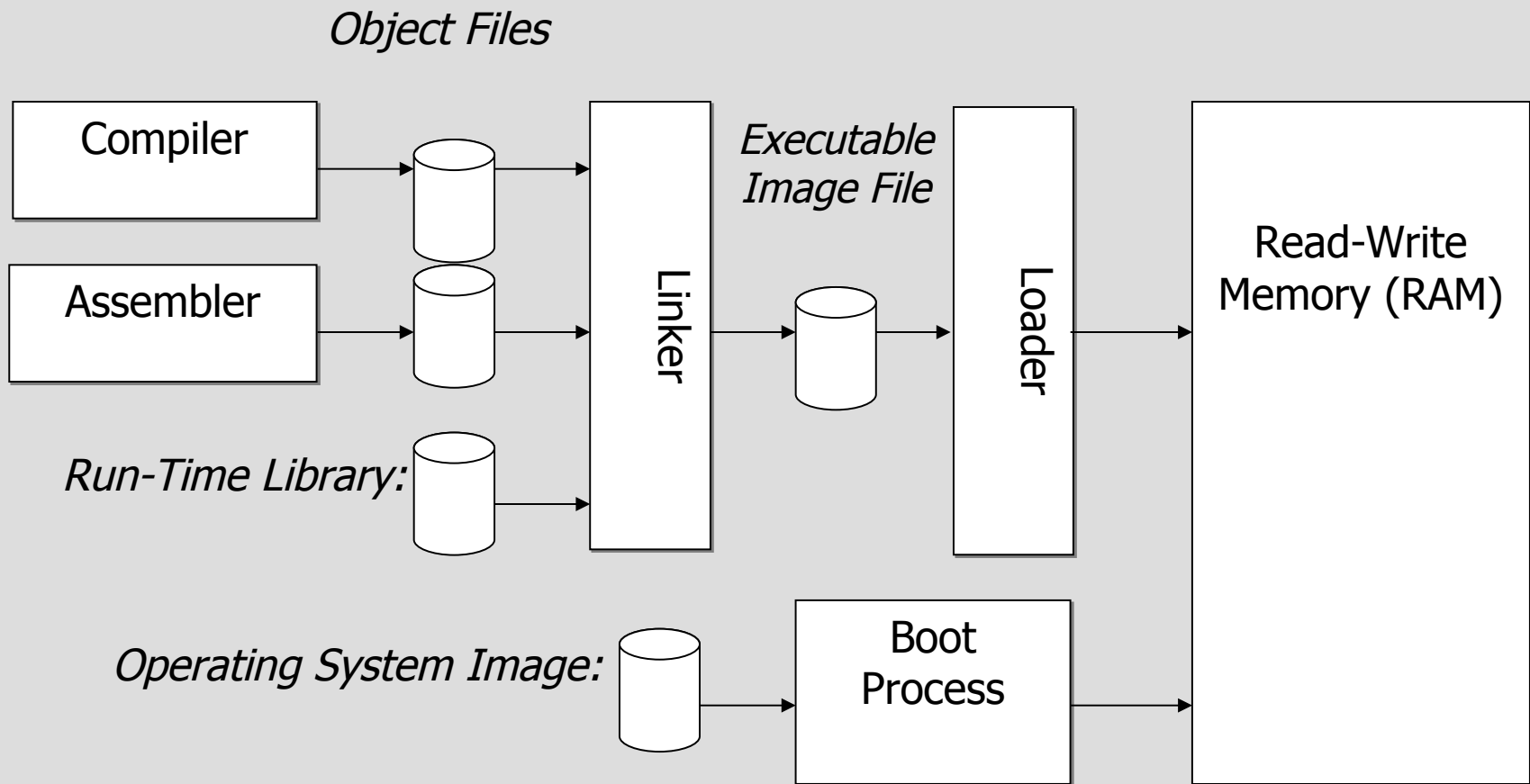
Objetivos

- Proceso de compilación de una aplicación
 - Compilación, linker/locator
 - Diferencias entre compiladores nativos y embebidos (cruzados).
- Inicialización de datos y arranque de programa
- Cargado de programa

Herramientas de desarrollo

- Conceptos (repaso):
 - *tool chain*
 - herramientas (compilador, ensamblador, etc.) compatibles
 - herramientas nativas (*native tools*)
 - Crea programas que corren en la computadora que se compila.
 - *host* (donde se desarrolla) vs. *target* (destino)
 - “compilador cruzado” (cross-compiler)
 - Compilador en *host* produce instrucciones binarias *target*.

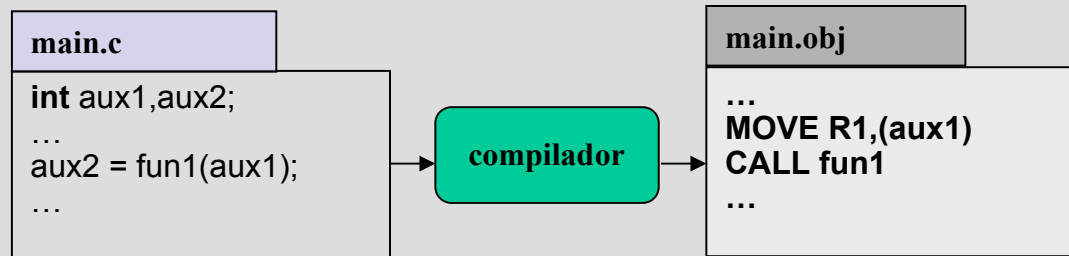
Proceso de creación y cargado: programa de aplicación de escritorio



Fuente: Daniel W. Lewis, "Fundamentals of Embedded Software"

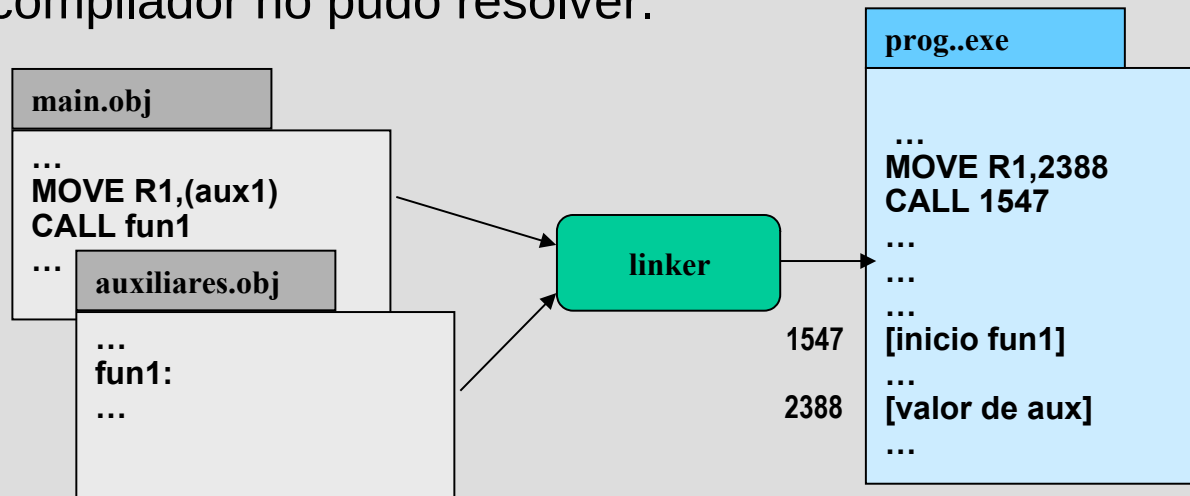
Compilador

- Traduce código fuente en lenguaje C a instrucciones de máquina generando un archivo objeto (unidad de compilación)



Enlazador (Linker)

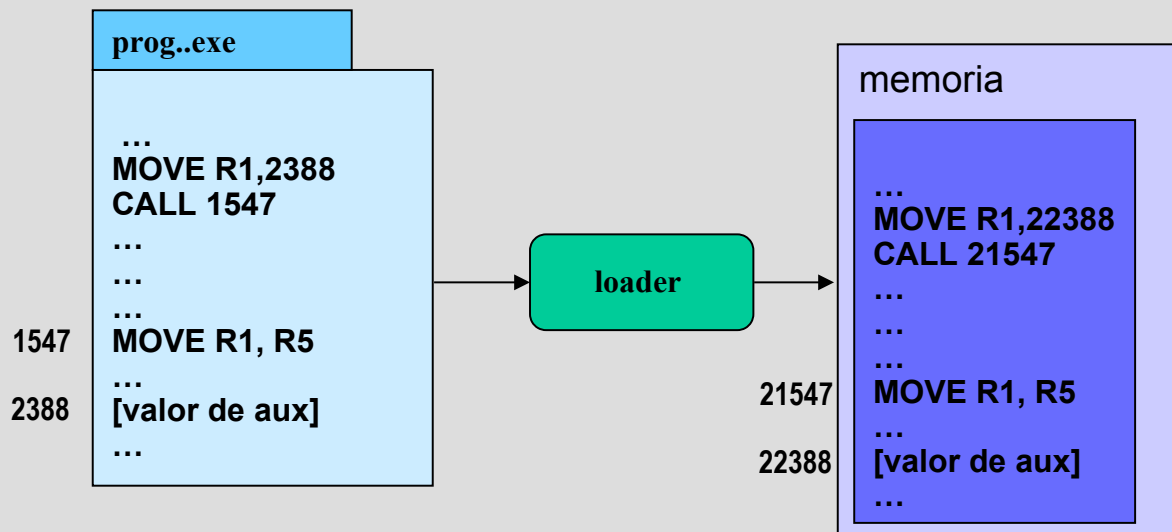
- “Junta” todos los archivos, resolviendo las referencias entre los archivos objetos
 - Determina las direcciones de las etiquetas que el compilador no pudo resolver.



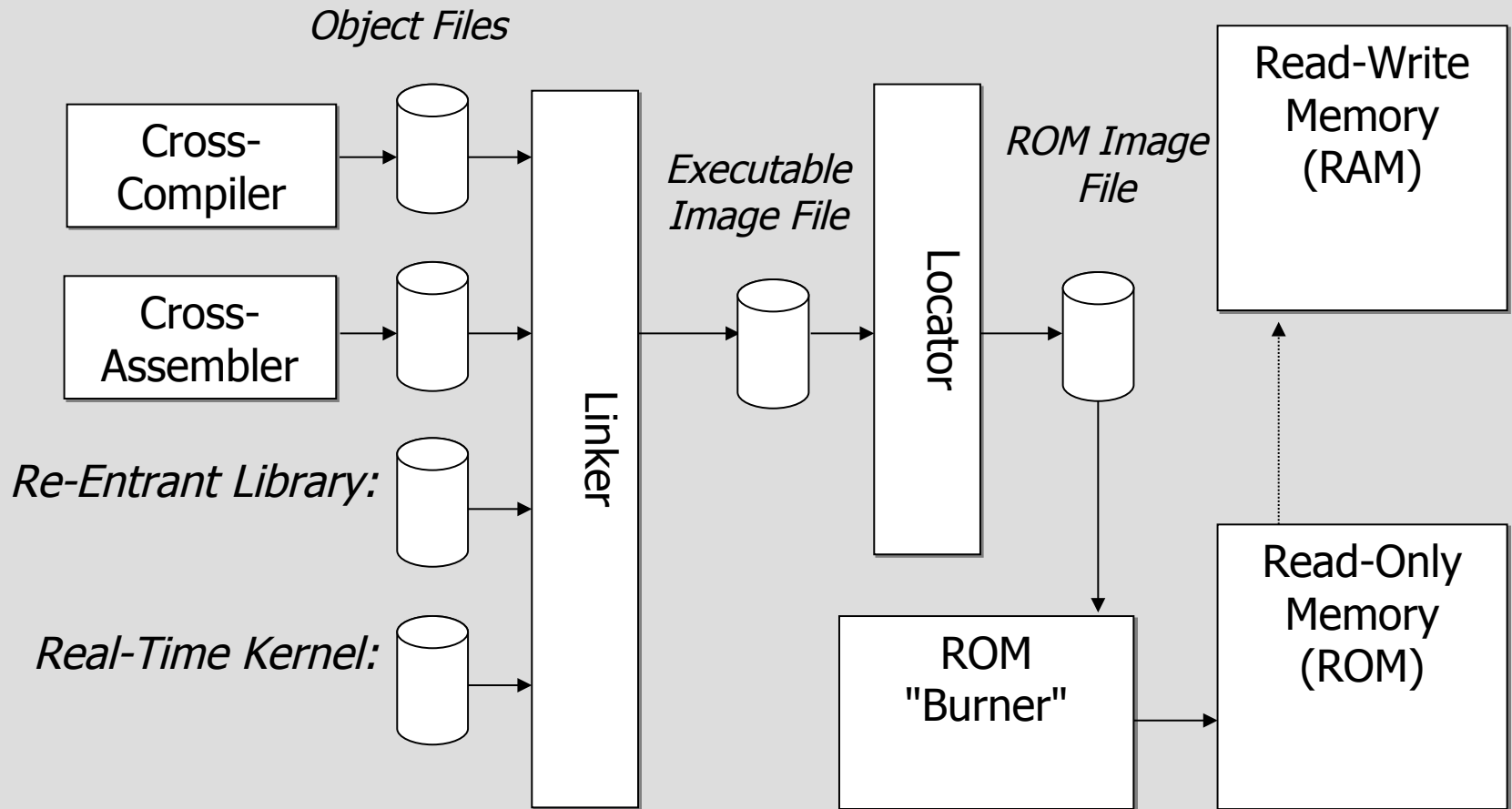
- Por ejemplo: funciones y variables externas (**extern**) definidas en otros archivos.

Loader

- Ajusta las direcciones de memoria absolutas del código y de los datos en función de la dirección de base
 - Obs: cada vez puede ser cargado en diferentes regiones de memoria



Proceso de creación y cargado: programa de aplicación embebido



Fuente: Daniel W. Lewis, "Fundamentals of Embedded Software"

Linker → Archivos Objeto

- ELF: Executable and Linking Format
 - Formato de código objeto ampliamente usado.
- Tipos de archivo objeto:
 - *ejecutable (executable file)*:
 - describe el programa para su ejecución
 - *objeto compartido (shared object file)*:
 - describe código y datos para *static* y *dynamic linking*
 - *relocalizable (relocatable file)*:
 - describe código y datos para *linking* con otros archivos, para crear un ejecutable o un objeto compartido.

(NO) Portabilidad de binarios

- Intrínseco de la compilación a código máquina (directamente ejecutable):
 - compiladores nativos y compiladores-cruzados
- Ejemplo de compilador nativo:
 - Pensar que un programa en C compila en cierta plataforma (hardware y sistema operativo).
 - ¿Qué problemas se tienen al cambiar de plataforma?
- Ejemplo de compiladores-cruzados:
 - Pensar que se compila para un microcontrolador dado...
 - ¿Qué problemas se tienen al cambiar de microcontrolador?

Locator

- Funcionalidad muy distinta a la del *loader*
 - Determina la imagen final en memoria del programa
 - No existe un proceso similar al correspondiente del *loader*
 - Es posible crear la imagen final porque:
 - Es el único programa en memoria, no existen conflictos por recursos.
 - Es posible determinar la dirección final de todo, incluyendo el kernel (si lo hubiera) y las funciones de biblioteca.
- Es posible indicar la ubicación en memoria
 - Datos, código.
 - RAM, ROM

Memorias (breve paréntesis)

- Volatilidad: incapacidad de retener data sin energía
 - Ejemplos:
 - Volátil: DRAM, SRAM.
 - No volátil: ROM, PROM, EPROM, EEPROM, Flash, y más recientemente MRAM, FeRAM o F-RAM.
- Read-write: capacidad de lectura/escritura
 - Ejemplos:
 - Read-only: ROM, PROM
 - Read-write: SRAM, DRAM
 - Depende: Flash, FeRAM
- Acceso
 - Ejemplos
 - Random-access memory (RAM): acceso aleatorio a posiciones arbitrarias (igual tiempo de acceso)
 - Secuencial

Actividad en grupo

- Inicialización de variables estáticas
 - Actividad:
 - 1) ¿cuándo se inicializan las variables estáticas?
 - 2) Reflexionar y describir (en palabras) la ejecución del código desde un reset
(se puede tomar como ejemplo el código solución de la slide siguiente)
 - Grupos:
 - 3 a 5 estudiantes
 - Tiempo:
 - 5 minutos

Modularización (solución actividad en slides de C)

com.h

```
void set_default_channel(int ch);
void load_default_channel();
void set_channel(int ch);
int read_channel();
```

main.c

```
#include "com.h"
#define N 10
int buf[N];

int main (void){
    int i;
    int channel=1;
    set_default_channel(channel);
    set_channel(2);
    for(i=1;i<N;i++){
        buf[i] = read_channel();
    }
}
```

com.c

```
#include "com.h"

static int default_channel = 1;
static int current_channel;

void set_default_channel(int ch){
    default_channel = ch;
}

void load_default_channel(){
    current_channel = default_channel;
}

void set_channel(int ch){
    current_channel = ch;
}

int read_channel(){
    current_channel = default_channel;
    return current_channel
}
```

Inicialización de datos

- Requerimientos:
 - Variable (modificable en ejecución) → read-write
 - Valor inicial (persistente entre reset) → no volátil
- Problema:
 - Valor inicial debe ser copiado al arranque (*startup*)
 - No hay *loader* → la aplicación debe hacerlo
- Solución:
 - *Linker/Locator* → automáticamente incluye código de startup
 - Startup code: copia valores iniciales (“shadow”) de mem no volátil (ej. Flash) a read-write (ej. SRAM)

Inicialización

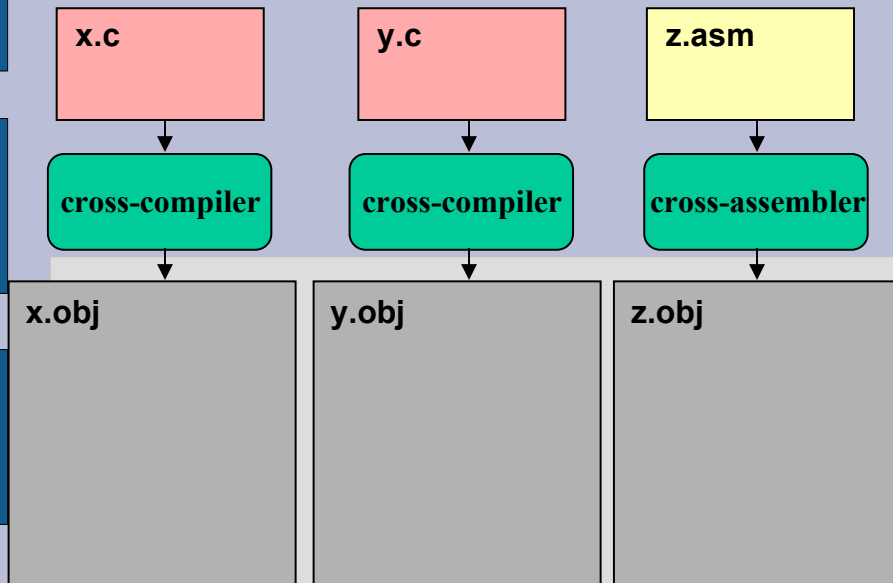
- Lenguaje C especifica sobre variables estáticas no inicializadas:

```
int i; //se debe inicializar en cero
```

- Requerimientos
 - variable (modificable) → read-write
 - valor inicial → cero
- Solución:
 - Se realiza un “memset” a cero (startup code)
- Startup code:
 - Inicialización de hardware
 - Inicialización de variables estáticas (en cero o valor inicial)
 - Llamado del main

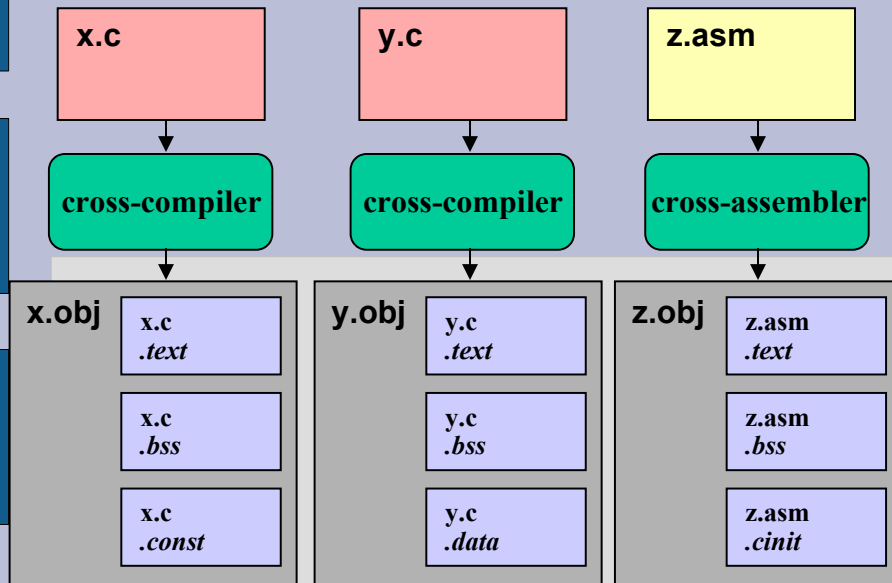
Ubicación de datos en memoria

- Tener en cuenta:
 - Datos persistentes necesitan estar en NVM (non-volatile memory).
 - Datos modificables necesitan estar en RAM.
- ¿Como sabe el linker/locator dónde poner cada cosa?
 - El programa se organiza en *segmentos* que son tratados independientemente.
- Segmentos
 - Compilador: crea entidades lógicas conteniendo código o datos
 - Linker: agrupa segmentos de diferentes códigos objeto.
 - Locator: mapea los segmentos en una ubicación física de memoria



Ejemplo

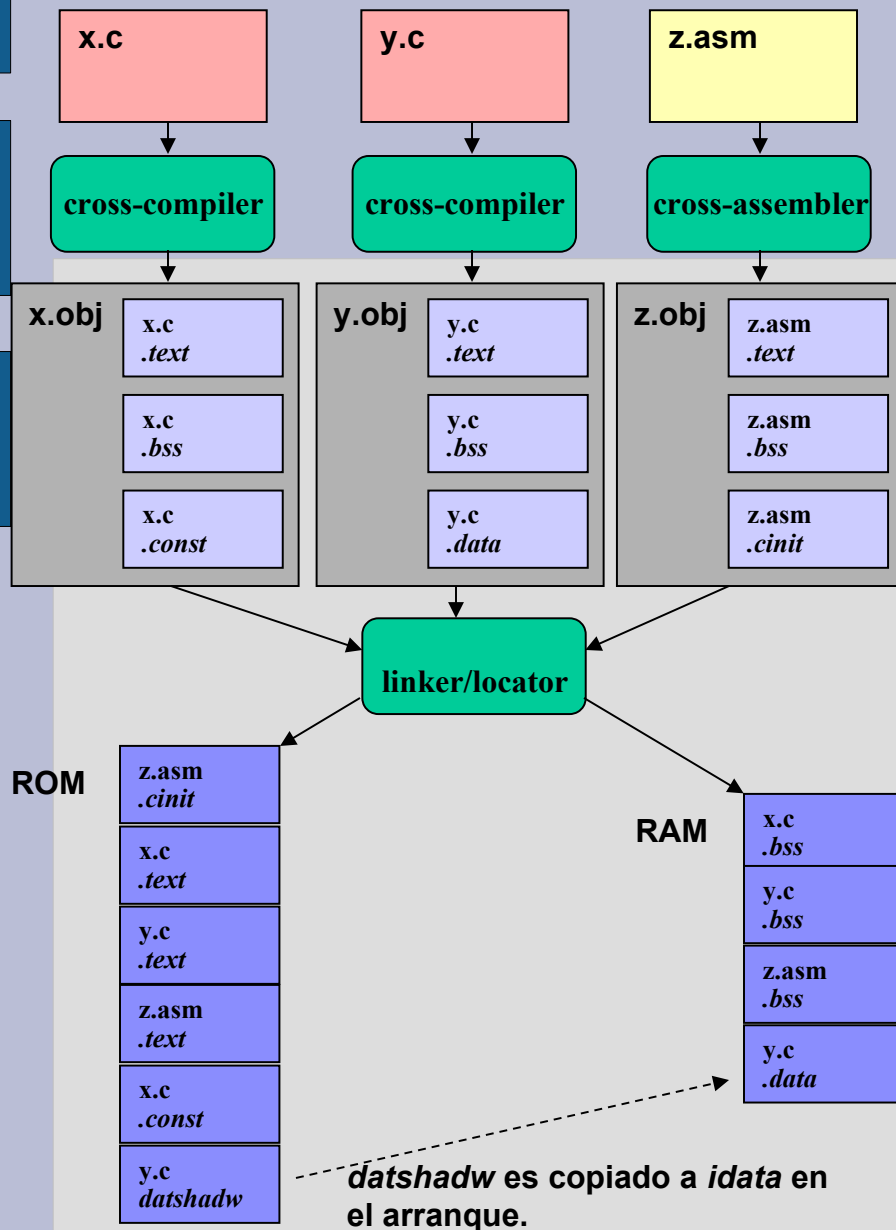
- Módulos:
 - x.c
 - Instrucciones
 - Datos sin inicializar
 - Constantes strings.
 - y.c
 - Instrucciones
 - Datos inicializados y sin inicializar
 - z.asm
 - Funciones en asm
 - Datos sin inicializar
 - Código de arranque (start-up code)



Ejemplo

- Cross-compiler
 - divide archivo fuente en segmentos
 - directivas al cross-comp
 - `#pragma`
- Segmentos:
 - **.text**: código ejecutable
 - **.bss**: datos sin inicializar
 - **.data**: datos inicializados
 - **.const**: constantes (incluye strings)
 - **.cinit**: código de arranque (start-up)
 - **.stack**: stack
 - Hay otros segmentos...

Ejemplo



- Segmentos

- **.text**: código ejecutable
- **.bss**: datos sin inicializar
- **.data**: datos inicializados
- **.const**: constantes (incluye strings)
- **.cinit**: código de arranque (start-up)

- Linker/locator

- “compagina” segmentos
- **cinit** copia **datshadw** a **.data** en el arranque, así tengo en RAM las variables siempre inicializadas en el mismo valor

Locator maps

- Mapa:
 - Salida adicional del *locator*
 - Usualmente en archivo *.map
- Resumen:
 - Ubicación del código, variables, etc. en los diferentes segmentos.
- Vemos juntos el locator.map generado por el CCS para un MSP430

Ejemplo de otro compilador

IAR C/C++

- Tipos de segmentos
 - CODE: código ejecutable
 - CONST: datos ubicados en ROM
 - DATA: datos ubicados en RAM
- Nomenclatura: nombrebase_sufijo
 - Nombre base
 - Por ejemplo: DATA16 (MSP430), NEAR (AVR)
 - Sufijo (categoría):
 - Datos no inicializados: N
 - Datos inicializados a cero: Z
 - Datos inicializados non-cero: I
 - Inicializadores para el anterior: ID
 - Constantes: C

Locator maps

- Mapa:
 - salida adicional del *locator*
 - usualmente *.map
- Resumen:
 - ubicación del código, variables, etc. en los diferentes segmentos.

```
*****
*
*          CROSS REFERENCE
*
*****

Program entry at : 020A Relocatable, from module : ?cstart

*****
*
*          MODULE MAP
*
*****

DEFINED ABSOLUTE ENTRIES
*****

DEFINED ABSOLUTE ENTRIES
PROGRAM MODULE, NAME : ?ABS_ENTRY_MOD

Absolute parts
ENTRY          ADDRESS          REF BY
=====
HEAP SIZE      0050
STACK SIZE     0050
*****

FILE NAME : C:\temp\Debug\Obj\segment.r43
PROGRAM MODULE, NAME : segment

SEGMENTS IN THE MODULE
=====
DATA16 C
Relative segment, address: 025C - 0265 (0xa bytes), align: 0
Segment part 2.          Intra module refs:  str1
-----
DATA16 C
Relative segment, address: 025A - 025B (0x2 bytes), align: 1
Segment part 3.          Intra module refs:  main
ENTRY          ADDRESS          REF BY
=====
str1            025A
-----
DATA16 Z
Relative segment, address: 0200 - 0209 (0xa bytes), align: 0
Segment part 4.          Intra module refs:  main
ENTRY          ADDRESS          REF BY
=====
str2            0200
-----
CODE
Relative segment, address: 0222 - 023D (0x1c bytes), align: 1
Segment part 5.
ENTRY          ADDRESS          REF BY
=====
main           0222              Segment part 12
stack 1 = 00000000 ( 00000002 )
*****
```

Ejemplos

`ejemplo.c`

`...`

`...`

`int i;`

`int j = 4;`

`...`

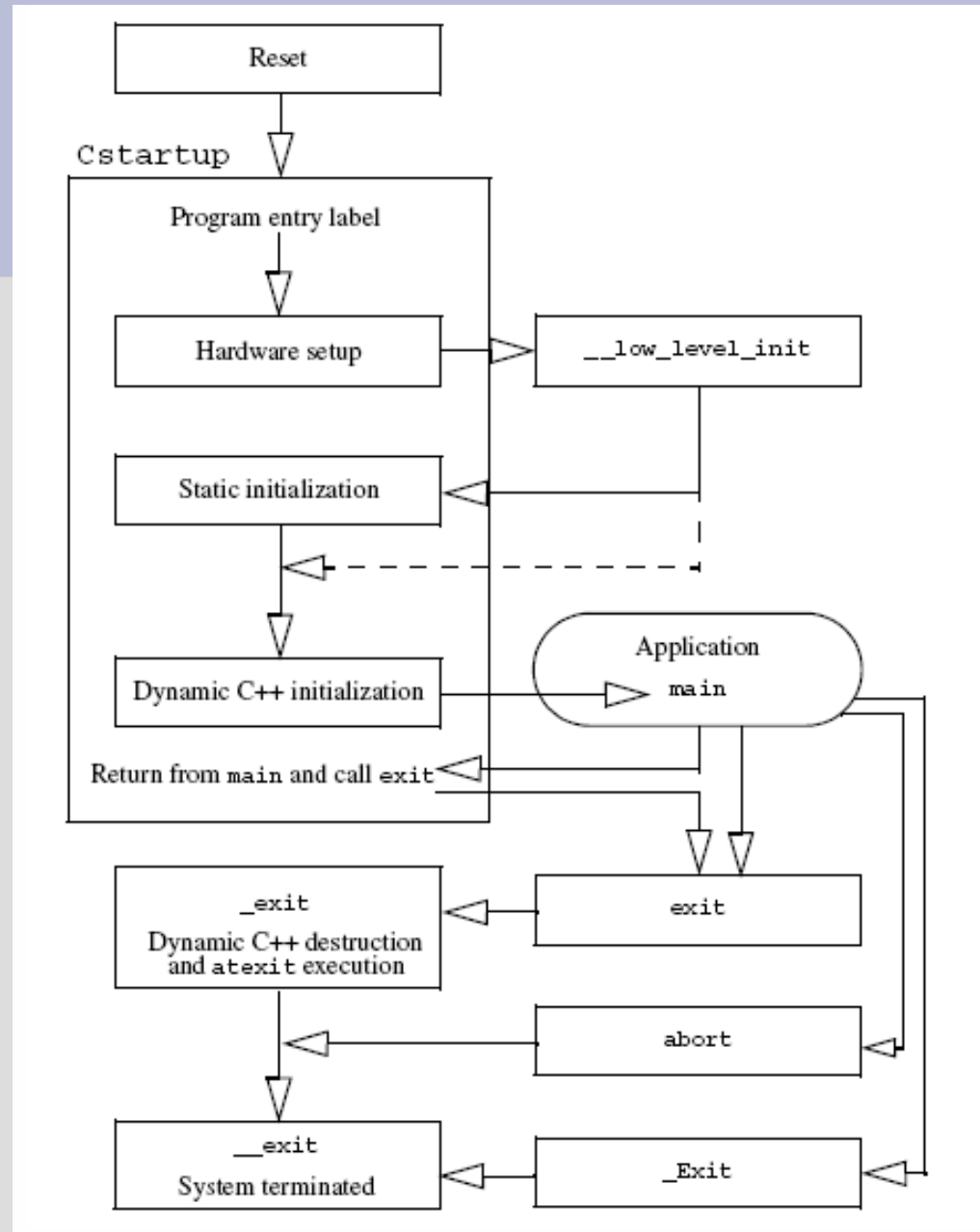
`...`

Variables que son inicializadas a cero son ubicadas en el segmento **DATA16_Z**

Variables inicializadas a valores distintos de cero son ubicadas en el segmento **DATA16_I** y un segmento de inicialización en **DATA16_ID**.

Compilador IAR C/C++

- Segmento CSTART contiene código para:
 - arranque (cstartup)
 - terminación (cexit).



Otros temas de inicialización

- Constantes strings:

```
char *sMsg = "Hola chau";
```

- ¿Dónde se guarda esta constante?
- ¿Qué pasa si modificamos este string?
 - Nota: es válido en C

- Compiladores pueden decir en función de:

- si es declarado constante:

```
char* const
```

- si es modificado o no en el código

Ejemplo

ej-string.c

```
char* const str1 = "Hola chau";
char str2[10];

int main( void )
{
    char* c1;
    char* c2;
    c1=str1;
    c2=str2;
    for (; *c1!=0; c1++, c2++)
        *c2 = *c1;
}
```

SEGMENTS IN THE MODULE

=====

DATA16_C

Relative segment, address: 025C - 0265 (0xa bytes), align: 0

Segment part 2.

Intra module refs:

str1

DATA16_C

Relative segment, address: 025A - 025B (0x2 bytes), align: 1

Segment part 3.

Intra module refs: main

ENTRY

ADDRESS

REF BY

str1

025A

DATA16_Z

Relative segment, address: 0200 - 0209 (0xa bytes), align: 0

Segment part 4.

Intra module refs: main

ENTRY

ADDRESS

REF BY

str2

0200

CODE

Relative segment, address: 0222 - 023D (0x1c bytes), align: 1

Segment part 5.

ENTRY

ADDRESS

REF BY

main

0222

Seg. part 12 (?cstart)

stack 1 = 00000000 (00000002)

Ejemplo

ej-string.c

```
char* str1 = "Hola chau";
char str2[10];

int main( void )
{
    char* c1;
    char* c2;
    c1=str1;
    c2=str2;
    for (; *c1!=0; c1++, c2++)
        *c2 = *c1;
}
```

SEGMENTS IN THE MODULE

=====

DATA16_I

Relative segment, address: 0200 - 0201 (0x2 bytes), align: 1

Segment part 3. Intra module refs: main

ENTRY	ADDRESS	REF BY
-------	---------	--------

str1	0200	
------	------	--

DATA16_Z

Relative segment, address: 0202 - 020B (0xa bytes), align: 0

Segment part 5. Intra module refs: main

ENTRY	ADDRESS	REF BY
-------	---------	--------

str2	0202	
------	------	--

CODE

Relative segment, address: 0236 - 0251 (0x1c bytes), align: 1

Segment part 6.

ENTRY	ADDRESS	REF BY
-------	---------	--------

main	0236	Seg. part 12 (?cstart)
------	------	------------------------

stack 1 = 00000000 (00000002)

DATA16_ID

Relative segment, address: 02A0 - 02A1 (0x2 bytes), align: 1

Segment part 4. Intra module refs: str1

DATA16_C

Relative segment, address: 0296 - 029F (0xa bytes), align: 0

Segment part 2. Intra module refs: Seg. part 4

Cargado del programa

- Alternativas (en orden cronológico de aparición):
 - Grabarlo en ROM o PROM, y después insertar el chip en la placa.
 - Grabarlo en la memoria FLASH destino.

Memoria FLASH

- Memoria FLASH:
 - no volátil y programable en campo
 - host puede conectarse al target y programarla
- Maneras:
 - BSL (*bootstrap program*)
 - programa que “sabe” guardar en FLASH el código que recibe por medio de un puerto de comunicación
 - JTAG (Joint Test Action Group)
 - puerto para debugging que permite el grabado en Flash

In-circuit emulator

- Origen del término: *emulator*
 - funcionalidad similar a un debugger de escritorio
 - Set breakpoints
 - Single-step
 - descarga de registros y memoria, etc
 - escritura en memoria
- Atención: ¡ya no es emulado!
- Ejemplo:
 - Flash Emulation Tool (FET): prototipado y desarrollo
 - Interfaz JTAG

Actualizaciones en campo

- Corregir código en campo es caro.
 - debe llevar el producto para actualizar el programa (memoria), eventualmente el cliente podría hacerlo.
- Algunos productos permiten actualización automática (OTA).
 - Receptores de TV Satélite o cable.
 - Teléfonos celulares
 - Motes (RSI)
- No es fácil:
 - Si la imagen obtenida está corrupta (problemas de comunicación), después no anda.

Bibliografía

- “An Embedded Software Primer” David Simon
 - Chapter 9: Embedded Software Development Tools
- “Real-time Embedded Systems” Xiaocong Fan
 - Chapter 2: Cross-Platform Development
- MSP430 Assembly Language Tools
v20.12.0.STS, User’s Guide