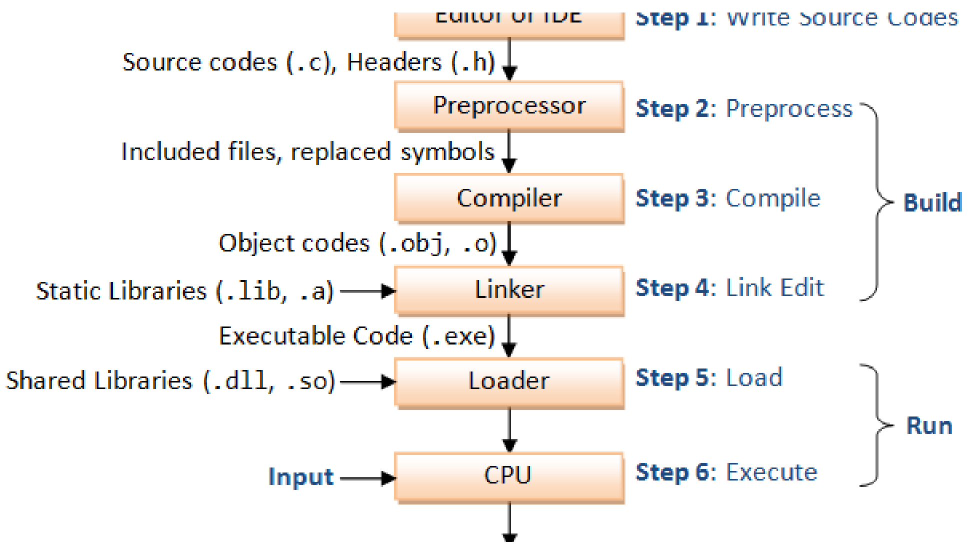


C Programming Compilation Process

Published on April 28, 2019

Ahmed Abd El-Ghafar Mohammed
Embedded Software Engineer at Avelabs

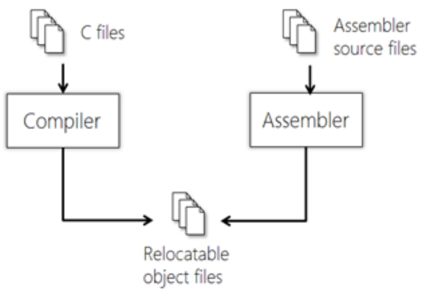
20 articles + Follow

In this article we look at the C build process.

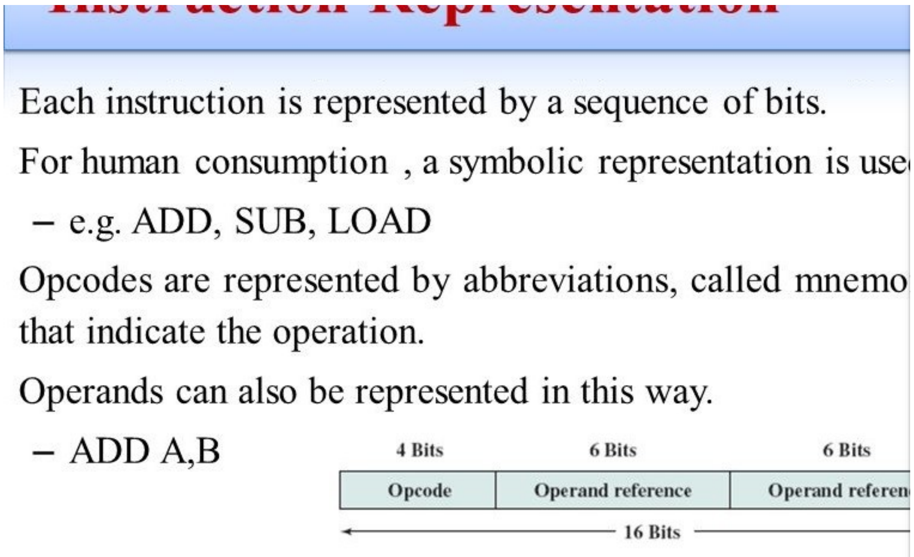
How we get from C source files to executable code, programmed on the target.

It wasn't so long ago this was common knowledge (the halcyon days of the hand-crafted make file!) but modern IDEs are making this knowledge ever-more arcane.

1- The first stage of the build process is "Compilation".



The compiler is responsible for allocating memory for definitions (static and automatic) and generating **Opcodes** (Unique binary number representing an operation to be carried out) from program statements.



Each instruction is represented by a sequence of bits.

For human consumption , a symbolic representation is use

– e.g. ADD, SUB, LOAD

Opcodes are represented by abbreviations, called mnemo that indicate the operation.

Operands can also be represented in this way.

– ADD A,B

4 Bits
6 Bits
6 Bits
Opcode
Operand reference
Operand referen
16 Bits

A **Relocatable Object** file (.o) is produced. The **assembler** also produces .o files fro assembly language source.

The compiler and assembler create relocatable object files (.o).

A **Librarian** facility may be used to take the object files and combine them into a library file.

The compiler works with one translation unit at a time (Takes .c file one by one).

A **translation unit** is a .c file that has passed through the **pre-processor**.

Compilation stages

Compilation is a multi-stage process, Each stage working with the output of the previous.

The Compiler itself is normally broken down into three parts :-

1. The **Front End**, responsible for (parsing the source code).
2. The **Middle End**, responsible for (optimization).
3. The **Back End**, responsible for (code generation).

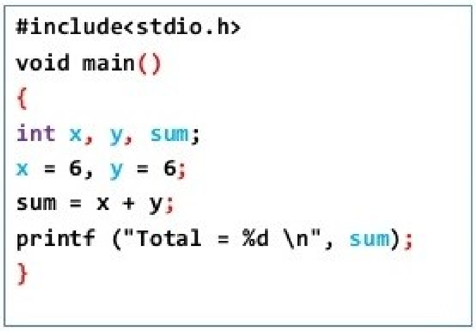
1 - Front End Processing :

- **Pre-processing :-** The pre-processor parses the source code file and evaluates pre-processor directives (starting with a #) – for example #define. A typical function of the pre-processor is to#include function / type declarations from header files.The input to the pre-processor is known as a ***pre-processed translation unit*** and the output from the pre-processor is a ***post-processed translation unit.***

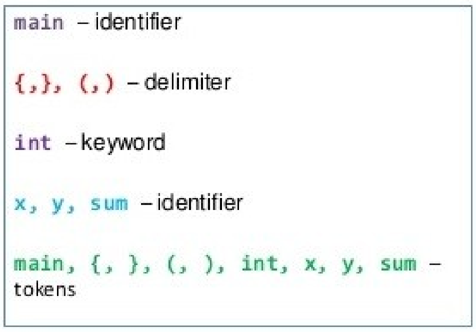
#undef identifier	Undefines a compilation symbol.
#if expression	If the expression is true, the compiler compiles the following section.
#elif expression	If the expression is true, the compiler compiles the following section.
#else	If the previous #if or #elif expression is false, the compiler compiles the following section.
#endif	Marks the end of an #if construct.
#region name	Marks the beginning of a region of code; has no compilation effect.
#endregion name	Marks the end of a region of code; has no compilation effect.
#warning message	Displays a compile-time warning message.
#error message	Displays a compile-time error message.
#line indicator	Changes the line numbers displayed in compiler messages.
#pragma text	Specifies information about the program context.

- **Whitespace removal :-** C ignores whitespace so the first stage of processing the translation unit is to strip out all whitespace.

- **Tokenizing :-** A C program is made up of *tokens and a* token may be :- a keyword (for example 'while'), an operator (for example, '*'), an identifier; a variable name, a literal (for example, 10 or "my string"), a comment (which is discarded at this point).



```
#include<stdio.h>
void main()
{
int x, y, sum;
x = 6, y = 6;
sum = x + y;
printf ("Total = %d \n", sum);
}
```



main – identifier

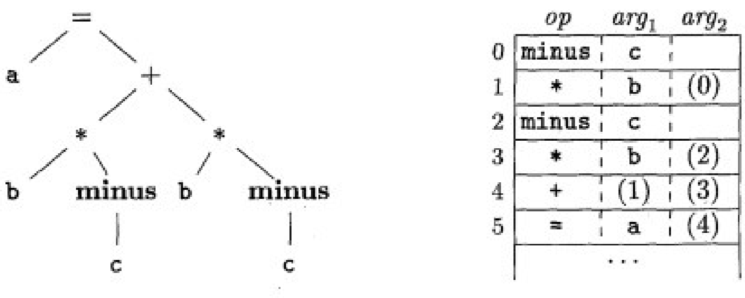
{,}, (,) – delimiter

int – keyword

x, y, sum – identifier

main, {, }, (,), int, x, y, sum – tokens

- **Syntax analysis :-** Syntax analysis ensures that tokens are organized in the correct way, according to the rules of the language. If not, the compiler will produce *a syntax error* at this point. The output of syntax analysis is a data structure known as **a *parse tree.***



	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
		...	

(a) Syntax tree (b) Triples

Figure 6.11: Representations of $a + a * (b - c) + (b - c) * d$

- **Intermediate Representation** :- The output from the compiler front end is a functionally equivalent program expressed in some *machine-independent form* known as an *Intermediate Representation* (IR). The IR program is generated from the parse tree. IR allows the compiler vendor to support multiple different languages (for example C and C++) on multiple targets without having *n * m* combinations of tool-chain. There

2 - Middle End Processing :

- **Semantic analysis :-** Semantic analysis adds further **semantic information** to the IR AST and performs checks on the **logical** structure of the program. The type and amount of semantic analysis performed varies from compiler to compiler but most modern compilers are able to detect potential problems such as unused variables, uninitialized variables, etc. Any problems found at this stage are normally presented as ***warnings***, rather than errors. It is normally at this stage the **program symbol table** is constructed, and any **debug information inserted**.

- **Optimization :-** Optimization transforms the code into a functionally-equivalent, but smaller or faster form. Optimization is usually a multi-level process. Common optimizations include in-line expansion of functions, dead code removal, loop unrolling, register allocation, etc.

3 - Back End Processing :

- **Code generation :-** Code generation converts the optimised IR code structure into native opcodes for the target platform.

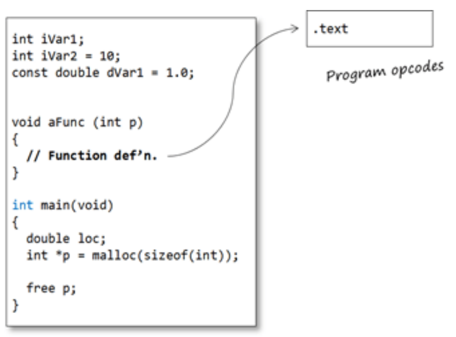
Memory allocation

The C compiler allocates memory for code and data in ***Sections***.

Each section contains a different type of information, Sections may be identified by name and/or with attributes that identify the type of information contained within.

This (**attribute information)** is used by the Linker for locating sections in memory as we see later.

- **Code (.text) Section :** Opcodes generated by the compiler are stored in their own memory section, typically known as .code or .text



```
int iVar1;
int iVar2 = 10;
const double dVar1 = 1.0;


void aFunc (int p)
{
  // Function def'n.
}

int main(void)
{
  double loc;
  int *p = malloc(sizeof(int));

  free p;
}
```

- **Static data :** The static data region is actually subdivided into two further sections

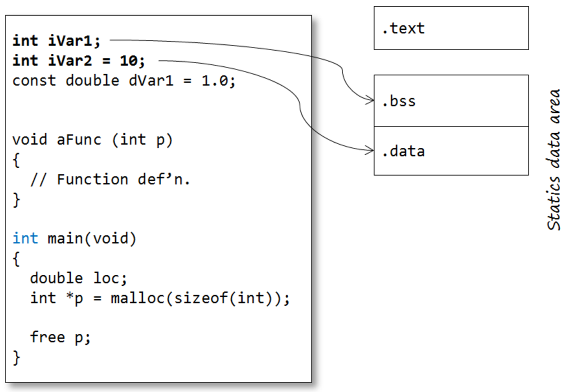
1. Uninitialized definitions, for ex. (int iVar1;)

2. Initialized definitions, for ex. (int iVar2 = 10;)

other in memory.

The uninitialized-definitions' section is commonly known as the (**.bss** or **ZI section)**.

The initialized definitions' section is commonly known as the **(.data or RW section)**.



```
int iVar1;
int iVar2 = 10;
const double dVar1 = 1.0;


void aFunc (int p)
{
  // Function def'n.
}

int main(void)
{
  double loc;
  int *p = malloc(sizeof(int));

  free p;
}
```

- **Constants :** Constants may come in two forms:

1. User-defined constant objects, for ex. (const int number;)

2. Literals, for ex. (Macro definitions or Strings)

The traditional C model places user-defined const objects in the .data section, along with non-const statics (so they may not be truly constant – this is why C disallows using constant integers to initialize arrays, for example).

Literals are commonly placed in the .text / .code section.

Most compilers will optimize numeric literals away and use their values directly where possible.

The C language provides several different kinds of constants: *integer constants* such as 10 and 0x1C, *floating constants* such as 1.0 and 6.022e+23, and *character constants* such as 'a' and '\x10', C also provides *string literals* such as "ouch!" and "\n".

Every literal has a type, It may not be obvious, and it may vary across platforms,but the standard does specify it precisely.

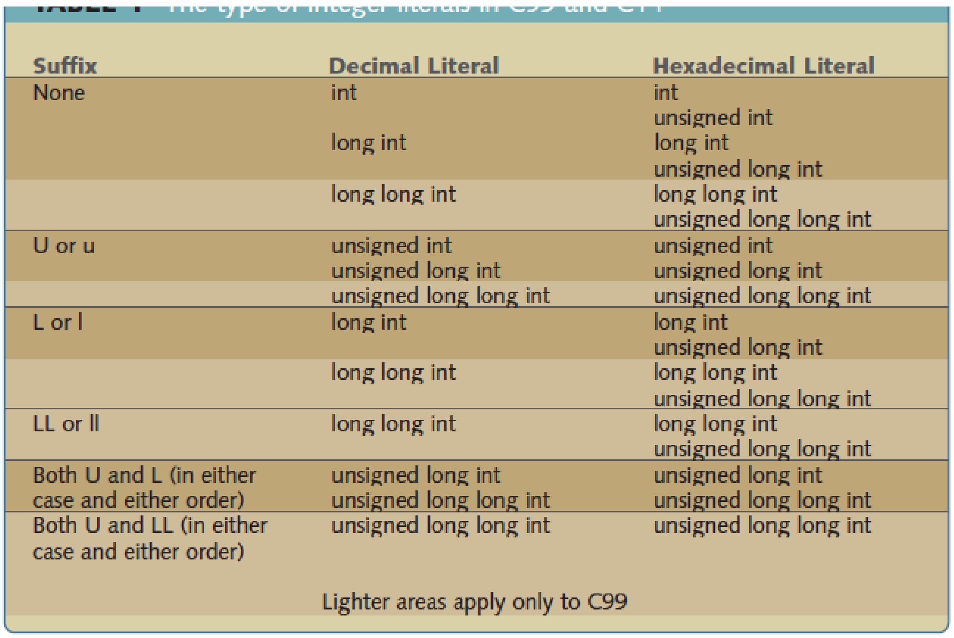


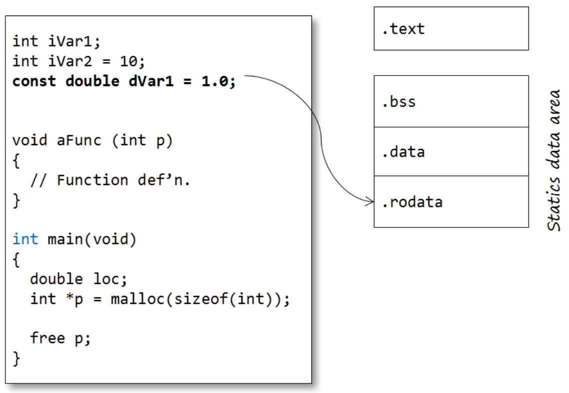
TABLE 1 The type of integer literals in C99 and C++

Suffix	Decimal Literal	Hexadecimal Literal
None	int	int unsigned int
	long int	long int unsigned long int
	long long int	long long int unsigned long long int
U or u	unsigned int	unsigned int
	unsigned long int	unsigned long int
	unsigned long long int	unsigned long long int
L or l	long int	long int unsigned long int
	long long int	long long int unsigned long long int
LL or ll	long long int	long long int unsigned long long int
Both U and L (in either case and either order)	unsigned long int unsigned long long int	unsigned long int unsigned long long int
Both U and LL (in either case and either order)	unsigned long long int	unsigned long long int

Lighter areas apply only to C99

Many modern C tool-chains support a separate **(.const / .rodata)** section specifically for constant values.

This section can be placed **(in ROM)** separate from the .data section.



```
int iVar1;
int iVar2 = 10;
const double dVar1 = 1.0;


void aFunc (int p)
{
  // Function def'n.
}

int main(void)
{
  double loc;
  int *p = malloc(sizeof(int));

  free p;
}
```

- **Automatic variables :** The majority of variables are defined within functions are classed as automatic variables, This also includes parameters and any temporary returned object (TRO) from a non-void function.

The default model in general programming is that the memory for these program objects is allocated from the stack and in this model, automatic memory is reclaimed by popping the stack on function exit.

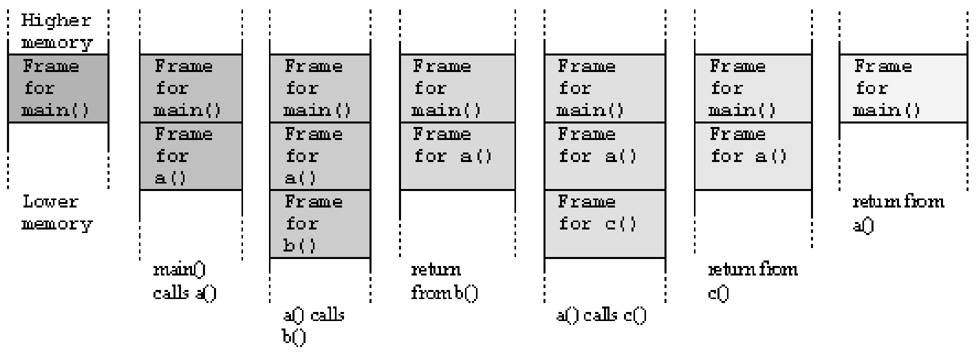
For parameters and TRO's the memory is normally allocated by the calling function (by pushing values onto the stack), whereas for local objects, memory is allocated once the function is called.

This key feature enables a function to call itself recursion (though recursion is generally a bad idea in embedded programming as it may cause (**stack overflow** problems).

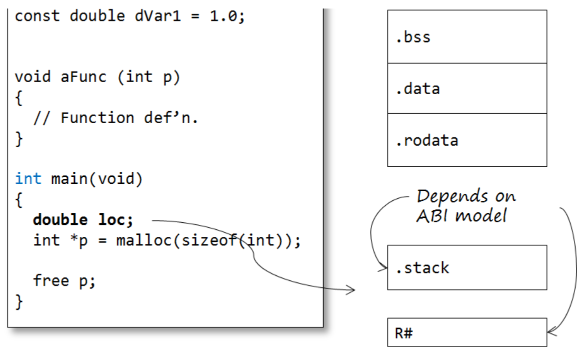
It is important to note that the compiler doesn't create a (**.stack**) segment, Instead, opcodes automatics are stored in scratch registers, where possible, rather than the stack.

Register	Synonym	Special	Role in the procedure call standard
r15	-	pc	Program counter.
r14	-	lr	Link register.
r13	-	sp	Stack pointer.
r12	-	ip	Intra-procedure-call scratch register.
r11	v8	-	ARM-state variable register 8.
r10	v7	sl	ARM-state variable register 7. Stack limit pointer in stack-checked variants.
r9	v6	sb	ARM-state variable register 6. Static base in RWPI variants.
r8	v5	-	ARM-state variable register 5.
r7	v4	-	Variable register 4.
r6	v3	-	Variable register 3.
r5	v2	-	Variable register 2.
r4	v1	-	Variable register 1.
r3	a4	-	Argument/result/scratch register 4.
r2	a3	-	Argument/result/scratch register 3.
r1	a2	-	Argument/result/scratch register 2.
r0	a1	-	Argument/result/scratch register 1.

Stack frames for every function call are pushed and pops from the STACK for every function call



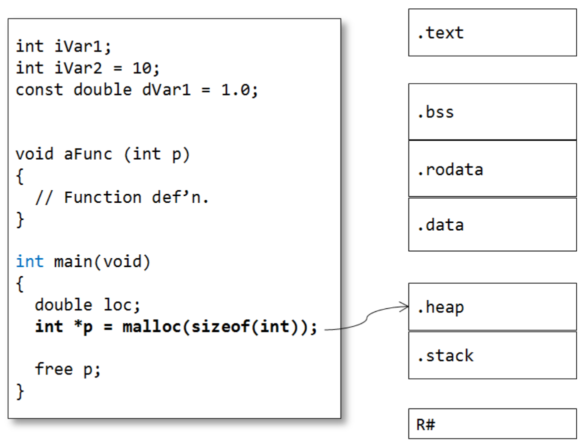
For example the ARM Architecture Procedure Call Standard (AAPCS) defines which CPU registers are used for function call arguments into, and results from, a function and local variables.



```
const double dVar1 = 1.0;


void aFunc (int p)
{
  // Function def’n.
}

int main(void)
{
  double loc;
  int *p = malloc(sizeof(int));

  free p;
}
```

- **Dynamic data :** Memory for dynamic objects is allocated from a section known as the(**Heap)**.

As with the Stack, the Heap is not allocated by the compiler at compile time but by **the Linker at link-time.**



```
int iVar1;
int iVar2 = 10;
const double dVar1 = 1.0;


void aFunc (int p)
{
  // Function def’n.
}

int main(void)
{
  double loc;
  int *p = malloc(sizeof(int));

  free p;
}
```

Object files

The compiler produces relocatable object files (.o) files, The object file contains the compiled **source code**, **opcodes** and **data sections**.

Note that the object file only contains the sections for static variables. At this stage, section locations are not fixed.

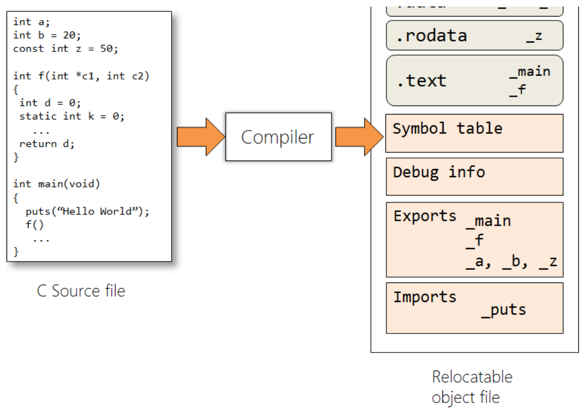
The .o file is not (yet) executable because, although some items are set in concrete (for example: instruction opcodes, pc-relative addresses, “immediate” constants, etc.), static and global addresses are known only as offsets from the starts of their relevant sections.

Also, addresses defined in other modules are not known at all, except by name.

The object file contains two tables - (Imports and Exports):

1. Exports contains any identifiers defined within this translation unit (so no statics!)

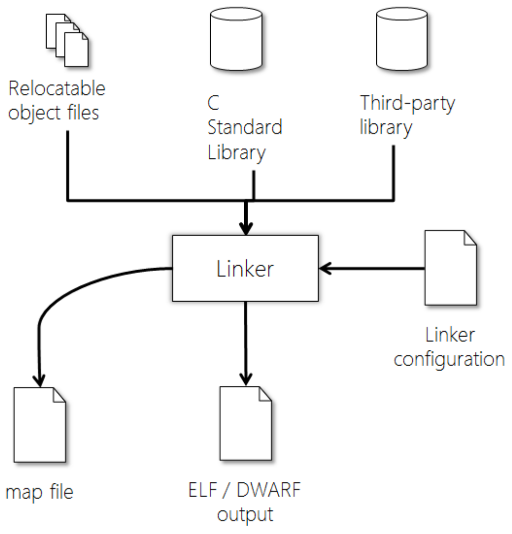
2. Imports contains any identifiers declared (and used) within the translation; but not defined within it.



```
int a;
int b = 20;
const int z = 50;

int f(int *c1, int c2)
{
 int d = 0;
 static int k = 0;
   ...
 return d;
}

int main(void)
{
  puts(“Hello World”);
  f()
   ...
}
```

Linking

The Linker combines the (compiled) object files into a single executable program and in order to do that it must perform a number of tasks.



- **Symbol resolution :** The primary function of the Linker (from whence it derives its name) is to resolve references between object files and to ensure each symbol defined by the program has a unique address.

If any references remain unresolved, all specified library/archive (.a) files are searched and the appropriate modules are gathered in order to resolve those references.

This is an iterative process. If, after this, the Linker still cannot resolve a symbol it will report an ‘**unresolved reference**’ **error**.

Similarly, C specifies a ‘one definition rule’ , each symbol must have a unique and *unambiguous* address.

If the Linker finds the same symbol defined in two object files it will report a ‘**redefinition**’ **error** (be careful, though – some older C compilers assume that the same symbol defined in two translation units must refer to the same object!)

- **Section concatenation :** The Linker then concatenates like-named sections from the sections. Program addresses are adjusted to take account of the concatenation.

- **Section location :** To be executable code all data sections must be located at absolute addresses in memory.

Each section is given an absolute address in memory. This can be done on a section-by-section basis but more commonly sections are concatenated from some base address.

Normally there is one base address in non-volatile memory for persistent sections (for example code) and one address in volatile memory for non-persistent sections (for example the Stack).

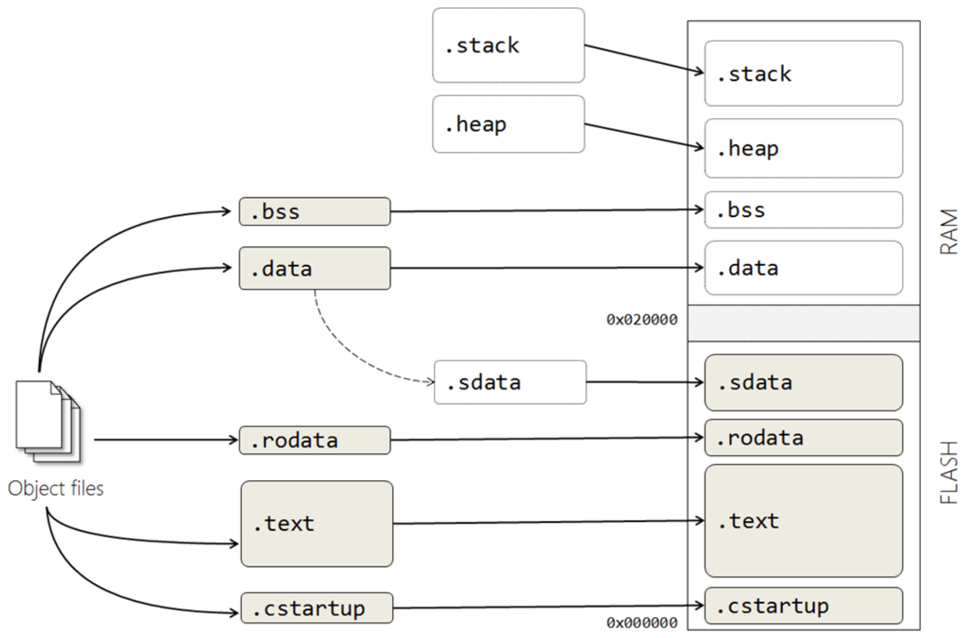
- **Data initialization:** On an embedded system any initialized data must be stored in non-volatile memory (Flash / ROM).

On startup any non-const data must be copied to RAM. It is also very common to copy read only sections like code to RAM to speed up execution.

In order to achieve this the Linker must create extra sections to enable copying from ROM to RAM.

Each section that is to be initialized by copying is divided in two parts, one for the ROM part (the initialization section) and one for the RAM part (the run-time location).

The initialization section generated by the **Linker** is commonly called a *shadow data* section (**.sdata**) in our example (although it may have other names).



1. The **.cstartup** – the system boot code – is explicitly located at the start of Flash.

2. The .**text** and .**rodata** are located in Flash, since they need to be persistent

3. The .**stack** and .**heap** are located in RAM.

is in ROM.

If manual initialization is not used, the linker also arranges for the (**startup code**) to perform the initialization.

The (**.bss**) section is also located in RAM but does *not* have a shadow copy in ROM.

A shadow copy is unnecessary, since the .bss section contains only zeros, This section can be initialized algorithmically as part of the startup code.

Linker control

The detailed operation of the linker can be controlled by invocation (command-line) options or by a ***Linker Control File*** (LCF) **(linker script file)**.

The linker script file file defines the physical memory layout (Flash/SRAM) and placement of the different program regions.

LCF syntax is highly **compiler dependent**, so each will have its own format, although the role performed by the LCF is largely the same in all cases.

Simple Linker Script Example :

The simplest possible linker script has just one command: 'SECTIONS', You use the 'SECTIONS' command to describe the memory layout of the output file.

The 'SECTIONS' command is a powerful command. Here we will describe a simple use of it. Let's assume your program consists only of code, initialized data, and uninitialized data. These will be in the '.text', '.data', and '.bss' sections, respectively.

Let's assume further that these are the only sections which appear in your input files.

For this example, let's say that the code should be loaded at address 0x10000, and that the data should start at address 0x8000000. Here is a linker script which will do that:

```
SECTIONS
{
  . = 0x10000;
  .text : { *(.text) }
  . = 0x8000000;
  .data : { *(.data) }
  .bss : { *(.bss) }
}
```

You write the 'SECTIONS' command as the keyword 'SECTIONS', followed by a series of symbol assignments and output section descriptions enclosed in curly braces.

The first line inside the 'SECTIONS' command of the above example sets the value of the start of the 'SECTIONS' command, the location counter has the value '0'.

The second line defines an output section, '.text'. The colon is required syntax which may be ignored for now. Within the curly braces after the output section name, you list the names of the input sections which should be placed into this output section. The '*' is a wild-card which matches any file name. The expression '*(.text)' means all '.text' input sections in all input files.

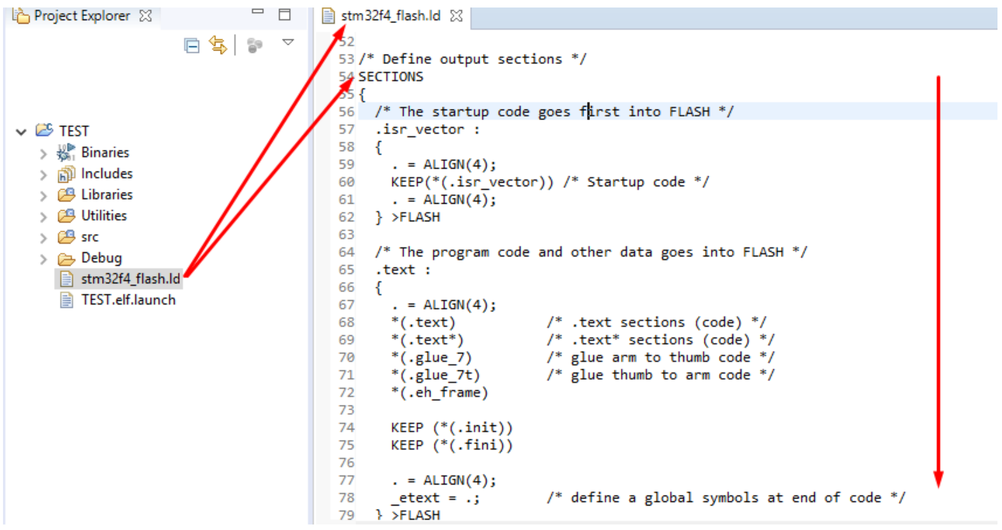
Since the location counter is '0x10000' when the output section '.text' is defined, the linker will set the address of the '.text' section in the output file to be '0x10000'.

The remaining lines define the '.data' and '.bss' sections in the output file. The linker will place the '.data' output section at address '0x8000000'. After the linker places the '.data' output section, the value of the location counter will be '0x8000000' plus the size of the '.data' output section. The effect is that the linker will place the '.bss' output section immediately after the '.data' output section in memory.

The linker will ensure that each output section has the required alignment, by increasing the location counter if necessary. In this example, the specified addresses for the '.text' and '.data' sections will probably satisfy any alignment constraints, but the linker may have to create a small gap between the '.data' and '.bss' sections.

When an IDE is used, these options can usually be specified in a relatively friendly way.

The IDE then generates the necessary script and invocation options.



```
/* Define output sections */
SECTIONS
{
  /* The startup code goes first into FLASH */
  .isr_vector :
  {
    . = ALIGN(4);
    KEEP(*(.isr_vector)) /* Startup code */
    . = ALIGN(4);
  } >FLASH

  /* The program code and other data goes into FLASH */
  .text :
  {
    . = ALIGN(4);
    *(.text)           /* .text sections (code) */
    *(.text*)          /* .text* sections (code) */
    *(.glue_7)         /* glue arm to thumb code */
    *(.glue_7t)        /* glue thumb to arm code */
    *(.eh_frame)

    KEEP (*(.init))
    KEEP (*(.fini))

    . = ALIGN(4);
    _etext = .;        /* define a global symbols at end of code */
  } >FLASH
```

The most important thing to control is where the final memory sections are located, The hardware memory layout must obviously be respected for most processors, certain things must be in specific places.

Secondly, the LCF specifies the size and location of the Stack and Heap (if dynamic memory is used). It is common practice to locate the Stack and Heap with the Heap at the lower address in RAM and the Stack at a higher address to minimize the potential for the two areas overlapping designated regions of memory.

The output from the locating process is a (**load file)** in a platform independent format, commonly .**ELF** or .**DWARF** (although there are many others).

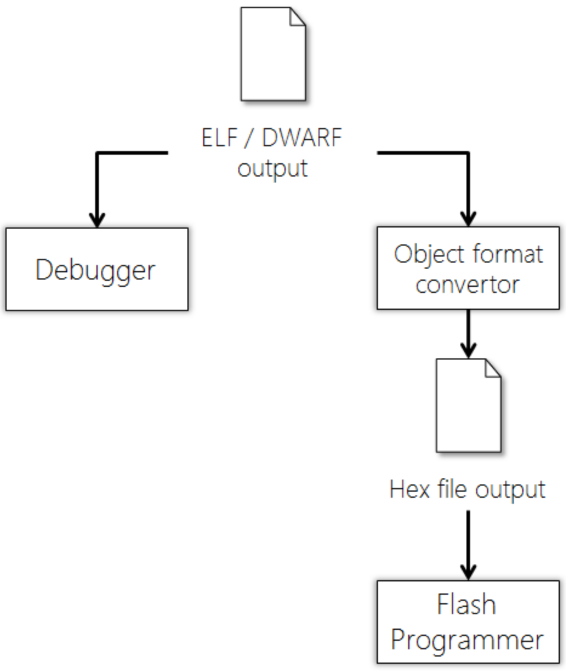
The ELF file is also used by the debugger when performing source code debugging.

Loading

ELF or DWARF are target independent output file formats. In order to be loaded onto the target the ELF file must be converted into a **native flash / PROM format** (typically, **.bin** or **.hex**)

```
:0400000038EF00F0E5
:1000080000C015F0F2A40DD0F2940B0ED76EDC0EE2
:10001800D66E1C4A1D2A1E4A1F2A204A212A1A4A1D
:100028001B2A9EA00DD09E900B0ECF6EDC0ECE6EBE
:100038001C4A1D2A1E4A1F2A204A212A1A4A1B2AFC
:0600480015C000F01100DC
:10004E00CD907F0ECD16CD9CCD82CF6ACE6ACD805F
:02005E0012008E
```

The Flash Programmer is then used to burn the code to the microcontroller :)



That is it :) I hope to share this article with others, Thanks :)

Report this

Reactions