A short course in linking and loading

John Kerl

October, 2003

Table of contents

1. A sample program

Here is a two-file C program (it doesn't do anything interesting):

```
// ----------------------------------------------------------------
// file1.c
// ----------------------------------------------------------------
int init1 = 3;
static int init2 = 4;
int uninit1;
static int uninit2;
char my_zstring[256];
char my_vstring[] = “Hello, world!”;
char * my_ptr = “How are you?”;

void main(void)
     // Embedded programs typically have nothing to return to,
     // hence return type void.
{
     int local1 = 3;
     int local2;

     local2 = 4;
     for (;;)
          func1(local1, local2);
}

// ----------------------------------------------------------------
// file2.c
// ----------------------------------------------------------------
int init3 = 17;
int uninit3;
char other_string[] = “Fine, thanks.”;
void func1(int arg1, int arg2)
{
     static int func1_static = 0;
     int * pdevreg = (int *)0xde0000ac;

     *pdevreg = func1_static;
     func1_static++;
}
```

This program demonstrates various types of memory. To see where it all goes as it is translated from C source code into an executable file to something that can run on a board, let's pretend to be the compiler, then the assembler, then the linker, then the code running on the board.

2. Hand-compiling the sample program

The compiler turns C source code into assembler. Here I'll use a fictitious assembly language.

Recall that the compiler sees one source file at a time, turning each one into an assembly file – e.g. `file1.c` to `file1.s`, `file2.c` to `file2.s` – and the assembler turns those into their corresponding object files – e.g. `file1.s` to `file1.o`, `file2.s` to `file2.o`. Then, *as a separate step*, the linker turns the object files into an executable file, e.g. `myprog`.

Before doing the assembly, I'll annotate the C code a little bit, to more clearly see what items go into which segments.

```
// file1.c
int init1 = 3;
```

Initialized global: `.data` *segment*

```
static int init2 = 4;
```

Initialized global: `.data` *segment*
`static` *for globals simply restricts the scope from program scope to file scope.*

```
int uninit1;
```

Uninitialized global: `.bss` *segment*

```
static int uninit2;
```

Uninitalized global (file scope): `.bss` *segment*

```
char my_zstring[256];
```

Uninitialized global: `.bss` *segment*

```
char my_vstring[] = “Hello, world!”;
char * my_ptr = “How are you?”;
```

Key point: In C, these two are different. `my_vstring` *is an array of characters, with length unspecified in the brackets. The length is taken from the initializer, which is 13 characters for* `“Hello, world!”` *plus the null string-termination character. Also, the ANSI C standard specifies that* `“Hello, world!”` *is the initial value, but that these values could be later modified at run time. Hence* `my_vstring` *is 14 bytes of read-write, initialized data. It goes in the* `.data` *segment.*

By contrast, `“How are you?”` *is a string literal, hence read-only. This is 12 characters, plus the null terminator. So, this string is 13 bytes in the* `.rodata` *segment.*

`my_ptr` *is a 4-byte pointer to character, whose initial value is the address of the string literal* `“How are you?”`*. However, that pointer could later, at run time, be assigned to point to something else. So,* `my_ptr` *is 4 bytes of read-write, initialized data. It goes into the* `.data` *segment.*

```
extern void func1(int, int);
```

This is just a prototype, to help the compiler do error checking. This statement generates no code.

```
void main(void)
```

This routine, `main()`*, is instructions, so it is in the* `.text` *segment.*

```
{
       int local1 = 3;
```

This is an initialized stack variable.

```
       int local2;
```

This is an uninitialized stack variable.

```
       local2 = 4;
       for (;;)
              func1(local1, local2);
```

This is a call to a function that is in another file

```
}

// file2.c
int init3 = 17;
```

This is an initialized global. `.data` *segment.*

```
int uninit3;
```

Uninitialized global. `.bss` *segment.*

```
char other_string[] = “Fine,
thanks.”;
```

As with `my_vstring` *in* `file1.c`*, this is an initialized global.* `.data` *segment.*

```
void func1(int arg1, int arg2)
{
```

This routine, `func1()`*, is instructions, so it is in the* `.text` *segment.*

```
       static int func1_static = 0;
```

The `static` *keyword in a function is different from* `static` *outside a function: Here, it means that the variable's value is retained between calls. This is still a global, even though only this function is allowed to refer to it by name – a rule which the compiler enforces. This variable is initialized to a specific value (even though that value is 0), so it goes into the* `.data` *segment.*

```
       int * pdevreg =
              (int *)0xde0000ac;
```

This is an initialized global (four bytes of pointer to integer, with an initial value specified) so it goes into the `.data` *segment.*

This idiom comes up a lot in embedded programming, and seldom or never when you write code that runs within an operating system: We know ahead of time that a certain device appears in the processor's memory space at a fixed address. Reading and/or writing to this address does some sort of device control. Let's suppose, for the sake of discussion, that eight data pins are wired somehow to eight LEDs, so that writing, for example, the byte 0xe0 to this address will turn on the first three LEDs and darken the remaining five.

```
       *pdevreg = func1_static;
```

Write a value to the LED device.

```
       func1_static++;
```

Increment for next call.

```
}
```

Now that we've analyzed the source code a little bit, we can pretend we're the compiler. Writing automated compilers isn't trivial, but for you and me (since we're human beings) the end result is pretty straightforward. Two key points, though, are (1) the compiler will put different things into different segments; (2) since the compiler sees each file one at a time, every object file has its own segments – `.text`, `.data`, `.bss`, etc. One of the linker's tasks is to shuffle all those segments together when it creates the executable file.

For the sake of discussion, suppose our fictitious processor has the following registers:

- `sp`, a stack pointer
- `pc`, a program counter
- `lr`, a link register
- `A`, an address register
- `D`, a data register
- `X` and `Y`, two more data registers

Here is the output of our hypothetical compiler. If you've printed this document on a black-and-white printer, you may be missing some color coding. I've color-coded as follows:

- `.data` items are green
- `.bss` items are orange
- `.text` items are blue
- `.rodata` items are red

```
# file1.s
# int init1 = 3;
       .segment data         init1 goes into file1.s's .data segment
       .export init1

init1:                       Ask the assembler to export the name init1 to the linker, so other
       .long 3               files can see this name.

# static int init2 = 4;
init2:
       .long 4               Also in file1.s's .data segment.  No export since it
                             has file scope.

# int uninit1;
       .segment bss          This goes in the .bss segment.
       .export uninit1       Export since it has program scope.
uninit1:
       .skip 4

# static int uninit2;        Also in .bss segment.  Has file scope, so no export.
uninit2:
       .skip 4

# char my_zstring[256];      Still in .bss segment
       .export my_zstring
my_zstring:
       .skip 256

# char my_vstring[] = "Hello, world!";
       .segment data         In .data segment
my_vstring:
       .ascii "Hello, world!", 0
       .align 4

# char * my_ptr = "How are you?";
       .export my_ptr        Pointer is in .data segment
my_ptr:
       .long lit001          Value is a symbolic name; address not known until link time.
```

```
        .segment rodata
lit001:                          This string literal goes into the .rodata segment.
        .ascii "How are you?", 0
        .align 4

# void main(void)                Code is in the .text segment.
# {
        .segment text
        .export main             Needs to be called by start(), in another file, so export it.
main:
        add  sp, 16
                                 main() uses two 4-byte local variables and two 4-byte arguments to its callee, func1():
                                 local1 is at sp+12 and local2 is at sp+16; outgoing argument 1 at sp+4 and
                                 argument 2 at sp+8.

#       int local1 = 3;          Assignment of stack variables happens at runtime.  The values
                                 are contained within the instructions in the .text segment.
                                 This is why the stack segment takes up no space in the executable file.
#       int local2;
        mov A, sp+12             Put address of local1 into register A.
        mov D, 3                 Put 3 into register D.
        st  D, A                 Store reg. D (value 3) back to address of local1.

#       local2 = 4;
        mov A, sp+16             Put address of local2 into register A.
        mov D, 4                 Put 4 into register D.
        st  D, A                 Store reg. D (value 4) back to address of local2.

L01:                             Compiler-generated symbol for top of loop.
#       for (;;)
#               func1(local1, local2);
        ld   D, sp+12            Marshal arguments for function call, passing arguments
        st   D, sp+4             by value (copy to new positions on stack).
        ld   D, sp+16
        st   D, sp+8
        bl func1
                                 Set pc to func1 (address not known till link time), saving address of next instruction in the
                                 link register (lr).
        b  L01                   Branch unconditionally to top of loop.
# }
        sub sp, 16               Restore context.
        ret                      Return to caller (address in lr).  Not reached due to for(;;).


# file2.s
# // file2.c
# int init3 = 17;
        .segment data            This goes into file2.s's .data segment.
        .export init3            Export, since it has global scope.
init3:
        .long 17

# int uninit3;                   This goes into file2.s's .bss segment.
        .segment bss
        .export uninit3          Export, since it has global scope.
uninit3:
        .skip 4

# char other_string[] = "Fine, thanks.";
        .segment data            This goes into file2.s's .data segment.
        .export other_string
```

```
other_string:
        .ascii "Fine, thanks.", 0

# void func1(int arg1, int arg2)
# {                                    Code goes into file2.s's .text segment.
        .segment text
        add sp, 4                      func1() uses one 4-byte local variable and calls no other function.
                                       Stack variable pdevreg is at sp+4.

#       static int func1_static = 0;
        .segment data                  Function statics are really function-scope globals.
func1::func1_static:
        .long 0

        .segment text                  Back in .text segment.
#       int * pdevreg = (int *)0xde0000ac;

        mov D, 0xde0000ac              Assignment of stack variables happens at runtime.
        st  D, sp+4                    Put 0xde0000ac at address of pdevreg (sp+4).

#       *pdevreg = func1_static;
        mov A, func1_static
        ld  D, A                       Copy data from address of global func1_static
        st  D, sp+4                    to address contained in stack variable pdevreg.

#       func1_static++;
        mov A, func1_static
        ld  D, A                       Load global variable to register, increment, store back.
        add D, 1
        st  D, A
# }
        sub sp, 4                      Restore context
        ret                            Return to caller
```

OK, so that's pretty simple – we just walk through the source code, assigning each statement to the segment in which it belongs. Roughly speaking, variables on the right-hand side of an equals sign (*rvalues* in compiler speak) turn into load instructions; variables on the left-hand side of an equals sign (*lvalues* in compiler speak) turn into store instructions.

3. Hand-assembling the sample program

Now, we'll pretend we're the assember. Like compilers, assemblers are non-trivial. However, for you and me, with our intuitive human minds, it will all be straightforward. I'll interleave the assembler with the machine code output. (The machine codes are entirely fictional as well as nonsensical.)

Notice that symbols resolved at link time have values set to zero at this point. For example, `main` in `file1.o`'s text segment calls `func1`, but the address of `func1` isn't known yet.

As above, I've color-coded:

- `.data` items are green
- `.bss` items are orange
- `.text` items are blue
- `.rodata` items are red

`file1.o:`

(Start of `file1.o`*'s* `.data` *segment)*

```
0x0000:  00 00 00 03        variable  init1
0x0004:  00 00 00 04        variable  init2
0x0008:  48 65 6c 6c        'H' 'e' 'l' 'l'   variable  my_vstring
0x000c:  6f 2c 20 77        'o' ',' ' ' 'w'
0x0010:  6f 72 6c 64        'o' 'r' 'l' 'd'
0x0014:  21 00 00 00        '!'  (null terminator) (two more 0 bytes for 4-byte alignment)
0x0018:  00 00 00 00        my_ptr.  After link, value will be address of lit001.
```

(Start of `file1.o`*'s* `.bss` *segment)*

```
0x001c:  00 00 00 00        variable  uninit1
0x0020:  00 00 00 00        variable  uninit2
0x0024:  00 00 00 00        First 4 bytes of  my_zstring
...      ...                248 more bytes of  my_zstring
0x0120:  00 00 00 00        Last 4 bytes of  my_zstring
```

(Start of `file1.o`*'s* `.rodata` *segment)*

```
0x0124:  48 6f 77 20        'H' 'o' 'w' ' '   variable lit001
0x0128:  61 72 65 20        'a' 'r' 'e' ' '
0x012c:  79 6f 75 3f        'y' 'o' 'u' '?'
0x0130:  00 00 00 00        (null terminator) (3 more 0 bytes for 4-byte alignment)
```

(Start of `file1.o`*'s* `.text` *segment)*

```
0x0134:  a8 9a 00 10        opcode for add sp, 16; start of main()
0x0138:  a9 80 04 0c        opcode for mov A, sp+12
0x013c:  a8 90 05 03        opcode for mov D, 3
0x0140:  a8 11 05 04        opcode for st  D, A
0x0144:  a9 80 04 10        opcode for mov A, sp+16
0x0148:  a8 90 05 04        opcode for mov D, 4
0x014c:  a8 11 05 04        opcode for st  D, A
0x0150:  a9 10 05 0c        opcode for ld  D, sp+12; label L01
0x0154:  a9 11 05 04        opcode for st  D, sp+4
0x0158:  a9 10 05 10        opcode for ld  D, sp+16
0x015c:  a9 11 05 08        opcode for st  D, sp+8
0x0160:  a8 31 00 00        opcode for bl func1
0x0164:  a8 30 00 00        opcode for b  L01
0x0168:  a8 9b 00 10        opcode for sub sp, 16
0x016c:  a8 40 01 00        opcode for ret
```

`file1.o`*'s symbol table (contained within* `file1.o`*):*

```
0x0000:  provide init1
0x0004:  provide init2
0x0008:  provide my_vstring
0x0018:  provide my_ptr
0x0018:  require lit001
0x001c:  provide uninit1
0x0020:  provide uninit2
0x0024:  provide my_zstring
0x0124:  provide lit001
0x0134:  provide main
0x0150:  provide L01
0x0160:  require func1
0x0164:  require L01
```

`file2.o`:

(Start of `file2.o`*'s* `.data` *segment)*

```
0x0000:  00 00 00 11        variable init3
0x0004:  46 69 6e 65        'F' 'i' 'n' 'e'; variable other_string
0x0008:  2c 20 74 68        ',' ' ' 't' 'h'
0x000c:  61 6e 6b 73        'a' 'n' 'k' 's'
0x0010:  2e 00 00 00        '.' (null terminator) (two more bytes for four-byte alignment)
0x0014:  00 00 00 00        variable func1_static, private to func1
```

(Start of `file2.o`*'s* `.bss` *segment)*

```
0x0018:  00 00 00 00        variable uninit3
```

(Start of `file2.o`*'s* `.text` *segment)*

```
0x001c:  a8 9a 00 04       opcode for add sp, 4; start of func1
0x0020:  a8 80 05 ff       opcode for mov D, imm
0x0024:  de 00 00 ac       immediate value for preceding mov
0x0028:  a9 11 05 04       opcode for st  D, sp+4
0x002c:  a8 80 04 ff       opcode for mov A, func1::func1_static
0x0030:  00 00 00 00       immediate value for preceding mov
0x0034:  a8 10 05 04       opcode for ld  D, A
0x0038:  a9 11 05 04       opcode for st  D, sp+4
0x003c:  a8 80 04 ff       opcode for mov A, func1::func1_static
0x0040:  00 00 00 00       immediate value for preceding mov
0x0044:  a8 10 05 04       opcode for ld  D, A
0x0048:  a8 91 05 01       opcode for add D, 1
0x004c:  a8 11 05 04       opcode for st  D, A
0x0050:  a8 9b 00 04       opcode for sub sp, 4
0x0054:  a8 40 01 00       opcode for ret
```

`file2.o`*'s symbol table (contained within* `file2.o`*):*

```
0x0000:  provide init3
0x0004:  provide other_string
0x0014:  provide func1::func1_static
0x0018:  provide uninit3
0x001c:  provide func1
0x0030:  require func1::func1_static
0x0040:  require func1::func1_static
```

4. Linker input map

In order to generate the executable file, the linker will need to assign segments to specific memory addresses. For programs running within an operating system, a default layout is used, of which the programmer is usually unaware. But for bare-board embedded systems, it is vital that the programmer tell the linker what goes where, typically using a *linker input map file*.

For this linking-and-loading example, let's assume the following:

- We are building a program which has read-write data, but is stored in flash.
- Earlier in this document, I talked about processor-init code. Let's suppose the C program we're building here executes out of the flash, but is independent from the processor-init code. (That is, the processor-init code will have already run, and then will simply jump into our program.)
- At runtime, the `.text` and `.rodata` segments will stay in flash.
- At runtime, the `.bss` segment will be in RAM and will need to be zero-filled.
- At runtime, the `.data` segment will need to be copied from its ROM storage location to its RAM location.

So, we will have the following expectations:

- The `.text` will go at a specified location in flash, say, 0xfff40000.
- The `.romdata` segment will go after the `.text` segment, in flash. (These are the initial values for the `.data` segment.)
- The `.rodata` segment will go after the `.romdata` segment, in flash.
- The `.data` segment will go at a specified location in SRAM, say, 0x10040000.
- The `.bss` segment will go after the `.data` segment, in SRAM.
- The stack will go at the end of the 1MB SRAM, with initial stack-pointer value 0x100ffff0.

How you tell the linker to do this depends entirely on your build tools. For the sake of discussion I'll use the following format:

```
_start 0xfff40000:
       .text:
       .romdata ROM(.data) align(4):
       .rodata:
.data 0x10040000:
       .bss:
.stack 0x100ffff0:
```

The idea is that a segment (or symbol name) with an address starts at that specified address; a segment (or symbol name) without an address starts where the previous region ended.

5. Library routines; _start

Input to the linker consists of the linker input map, plus all the user-specified object files, plus standard library files. (For example, typically you call `printf()` even though you didn't write it.) For simplicity, I made my little sample program use only one library routine (even though you might not have noticed): There is a function which calls `main()` – usually, it is named `_start`. In bare-board embedded systems, it usually doesn't do as much as it would in an operating-system environment, but still it must:

- Copy the `.data` segment from its ROM storage address to where it needs to go in RAM
- Zero-fill the `.bss` segment
- Set the stack pointer to the value specified in your linker input-map file
- Branch to `main`
- When (and if) `main` returns, either reset the processor or go into an infinite loop

Depending on your toolset, maybe you write `_start` yourself, or maybe it's a library routine. For the sake of discussion, let's assume that there's an assembly file that looks like this, named `crt0.s` (again, the name `crt0` is historical). Also, we'll suppose that as far as the source code is concerned, a segment named `.X` in the linker input map produces a pair of symbols `X_start` and `X_end`.

```
_start:

### Copy the .data segment from its ROM storage address to where it
### needs to go in RAM
       mov X, romdata_start       # X = source pointer
       mov Y, data_start          # Y = destination pointer
       mov A, romdata_end         # A = # bytes in .data segment
       sub A, romdata_start

data_copy:
       cmp A, 0                   # Byte counter down to 0 yet?
       bge data_copy_done
       ld  X, D                   # Load 32-bit word from .romdata segment
       st  D, Y                   # Store 32-bit word to data segment
       add X, 4                   # Increment .romdata pointer
       add Y, 4                   # Increment .data pointer
       sub A, 4                   # Decrement counter
       b data_copy                # Loop
data_copy_done:
```

```
### Zero-fill the .bss segment, 32 bits at a time:
        mov A, bss_start            # Address register = start of .bss
        mov D, bss_end              # Data register = # bytes in .bss
        sub D, bss_start
bss_fill:
        cmp D, 0                    # Counted down to 0 yet?
        bge bss_fill_done
        st  A,  0                   # Do a 32-bit write
        sub D,  4                   # Decrement the bytes-remaining counter
        add A,  4                   # Increment pointer that walks through .bss
        b bss_fill                  # Loop
bss_fill_done:

### Set the stack pointer to the value specified in the linker
### input-map file.
        mov sp, stack_start

### Branch to main.  In a bare-board embedded system, there is no
### argc nor argv to be passed.
        blr  main

### When (and if) main returns, go into an infinite loop.
spin:
        b spin
```

Since this library routine is used all the time, we'll suppose the cross-tools have it pre-assembled as `crt0.o`, which would look like this:

```
0x0000:  a8 80 05 ff  opcode for mov X, romdata_start; _start label
0x0004:  00 00 00 00  immediate value for preceding mov
0x0008:  a8 80 06 ff  opcode for mov Y, data_start
0x000c:  00 00 00 00  immediate value for preceding mov
0x0010:  a8 80 04 ff  opcode for mov A, romdata_end
0x0014:  00 00 00 00  immediate value for preceding mov
0x0018:  a8 82 04 ff  opcode for sub A, romdata_start
0x001c:  00 00 00 00  immediate value for preceding sub
0x0020:  a8 30 04 00  opcode for cmp A, 0; data_copy label
0x0024:  28 30 00 00  opcode for bge data_copy_done
0x0028:  a8 10 05 07  opcode for ld  X, D
0x002c:  a8 11 07 06  opcode for st  D, Y
0x0030:  a8 91 05 04  opcode for add X, 4
0x0034:  a8 91 06 04  opcode for add Y, 4
0x0038:  a8 82 04 04  opcode for sub A, 4
0x003c:  a8 30 00 00  opcode for b data_copy
0x0040:  a8 80 04 ff  opcode for mov A, bss_start; data_copy_done label
0x0044:  00 00 00 00  immediate value for preceding mov
0x0048:  a8 80 07 ff  opcode for mov D, bss_end
0x004c:  00 00 00 00  immediate value for preceding mov
0x0050:  a8 82 07 ff  opcode for sub D, bss_start
0x0054:  00 00 00 00  immediate value for preceding sub
0x0058:  a8 30 07 00  opcode for cmp D, 0; bss_fill label
0x005c:  28 30 00 00  opcode for bge bss_fill_done
0x0060:  a8 51 04 00  opcode for st  A,  0
0x0064:  a8 92 07 04  opcode for sub D,  4
0x0068:  a8 91 04 04  opcode for add A,  4
0x006c:  a8 30 00 00  opcode for b bss_fill
0x0070:  a8 80 02 ff  opcode for mov sp, stack_start; bss_file_done
0x0074:  00 00 00 00  immediate value for preceding mov
0x0078:  a8 31 00 00  opcode for blr  main
0x007c:  a8 30 00 00  opcode for b spin; spin label
```

`crt0.o`*'s symbol table (contained within* `crt0.o`*):*

```
0x0000:  provide _start
0x0004:  require romdata_start
0x000c:  require data_start
0x0014:  require romdata_end
0x001c:  require romdata_start
0x0020:  provide data_copy
0x0024:  require data_copy_done
0x003c:  require data_copy
0x0040:  provide data_copy_done
0x0044:  require bss_start
0x004c:  require bss_end
0x0054:  require bss_start
0x0058:  provide bss_fill
0x005c:  require bss_fill_done
0x006c:  require bss_fill
0x0070:  provide bss_file_done
0x0074:  require stack_start
0x0078:  require main
0x007c:  provide spin
0x007c:  require spin
```

6. Hand-linking the sample program

Now, we can pretend we're the linker, and link together `file1.o`, `file2.o and crt0.o`. As with compilers and assemblers, linkers are sophisticated technology, but you and I will easily be able to do this simple example by hand.

The linker needs to do the following:

- Put each input file's `.data` segments together into one big `.data` segment. Likewise for `.bss`, `.rodata` and `.text` segments. Each of these segments in the executable file will be contiguous blocks: The `.text` and `.rodata` segments, say, may or may not reside next to another at run time, but the `.text` segment itself won't be split up. Neither will any of the other segments.
- Resolve symbol references. Any time a symbol is required in an object file's symbol table, it must be provided exactly once, by one object file's symbol table. (Less than one `provide` yields an `undefined symbol` error; more than one `provide` yields a `multiply defined symbol` error.)
- Segments need to be assigned to specific memory addresses. For programs running within an operating system, a default layout is used, of which the programmer is usually unaware. But for bare-board embedded systems, it is vital that the programmer tell the linker what goes where, typically using a linker input map file. (See section 4, page 9, for more information on this topic.)
- Portions of the segments with unresolved references (currently filled with zeroes) need to be replaced with the correct values.
- The output needs to be written to a disk file, in one of several formats. (We'll discuss ELF and plain-binary formats.)

There are three layouts to be aware of:

- How the program will be stored on disk.
- How the program will be stored in flash.
- How the program will use memory at runtime.

6.1. Merging segments

The first link step is to merge like segments: our linker input map file specifies `_start` first at 0xfff40000, then `.rodata` after, then `.romdata` after that; discontiguously, `.data` at 0x10040000, then `.bss` after that. Given the color-coding I've used in this document, this just means to puts the blues together, then the reds, then the greens (duplicated – once for ROM, once for RAM), then orange. The first column in the object files had been file-relative offsets; now, they're adjusted to reflect the constraints in the linker input map file. These same adjustments will be applied to the contents of the symbol tables.

(Start of `crt0.o`*'s* `.text` *segment)*

```
0xfff40000:  a8 80 05 ff  opcode for mov X, romdata_start; _start label
0xfff40004:  00 00 00 00  immediate value for preceding mov
0xfff40008:  a8 80 06 ff  opcode for mov Y, data_start
0xfff4000c:  00 00 00 00  immediate value for preceding mov
0xfff40010:  a8 80 04 ff  opcode for mov A, romdata_end
0xfff40014:  00 00 00 00  immediate value for preceding mov
0xfff40018:  a8 82 04 ff  opcode for sub A, romdata_start
0xfff4001c:  00 00 00 00  immediate value for preceding sub
0xfff40020:  a8 30 04 00  opcode for cmp A, 0; data_copy label
0xfff40024:  28 30 00 00  opcode for bge data_copy_done
0xfff40028:  a8 10 05 07  opcode for ld  X, D
0xfff4002c:  a8 11 07 06  opcode for st  D, Y
0xfff40030:  a8 91 05 04  opcode for add X, 4
0xfff40034:  a8 91 06 04  opcode for add Y, 4
0xfff40038:  a8 82 04 04  opcode for sub A, 4
0xfff4003c:  a8 30 00 00  opcode for b data_copy
0xfff40040:  a8 80 04 ff  opcode for mov A, bss_start; data_copy_done label
0xfff40044:  00 00 00 00  immediate value for preceding mov
0xfff40048:  a8 80 07 ff  opcode for mov D, bss_end
0xfff4004c:  00 00 00 00  immediate value for preceding mov
0xfff40050:  a8 82 07 ff  opcode for sub D, bss_start
0xfff40054:  00 00 00 00  immediate value for preceding sub
0xfff40058:  a8 30 07 00  opcode for cmp D, 0; bss_fill label
0xfff4005c:  28 30 00 00  opcode for bge bss_fill_done
0xfff40060:  a8 51 04 00  opcode for st  A,  0
0xfff40064:  a8 92 07 04  opcode for sub D,  4
0xfff40068:  a8 91 04 04  opcode for add A,  4
0xfff4006c:  a8 30 00 00  opcode for b bss_fill
0xfff40070:  a8 80 02 ff  opcode for mov sp, stack_start; bss_file_done
0xfff40074:  00 00 00 00  immediate value for preceding mov
0xfff40078:  a8 31 00 00  opcode for blr  main
0xfff4007c:  a8 30 00 00  opcode for b spin; spin label
```

(Start of `file1.o`*'s* `.text` *segment)*

```
0xfff40080:  a8 9a 00 10  opcode for add sp, 16; start of main()
0xfff40084:  a9 80 04 0c  opcode for mov A, sp+12
0xfff40088:  a8 90 05 03  opcode for mov D, 3
0xfff4008c:  a8 11 05 04  opcode for st  D, A
0xfff40090:  a9 80 04 10  opcode for mov A, sp+16
0xfff40094:  a8 90 05 04  opcode for mov D, 4
0xfff40098:  a8 11 05 04  opcode for st  D, A
0xfff4009c:  a9 10 05 0c  opcode for ld  D, sp+12; label L01
0xfff400a0:  a9 11 05 04  opcode for st  D, sp+4
0xfff400a4:  a9 10 05 10  opcode for ld  D, sp+16
0xfff400a8:  a9 11 05 08  opcode for st  D, sp+8
0xfff400ac:  a8 31 00 00  opcode for bl func1
0xfff400b0:  a8 30 00 00  opcode for b  L01
0xfff400b4:  a8 9b 00 10  opcode for sub sp, 16
0xfff400b8:  a8 40 01 00  opcode for ret
```

(Start of `file2.o`*'s* `.text` *segment)*

```
0xfff400bc:  a8 9a 00 04  opcode for add sp, 4; start of func1
0xfff400c0:  a8 80 05 ff  opcode for mov D, imm
0xfff400c4:  de 00 00 ac  immediate value for preceding mov
0xfff400c8:  a9 11 05 04  opcode for st  D, sp+4
0xfff400cc:  a8 80 04 ff  opcode for mov A, func1::func1_static
0xfff400d0:  00 00 00 00  immediate value for preceding mov
0xfff400d4:  a8 10 05 04  opcode for ld  D, A
0xfff400d8:  a9 11 05 04  opcode for st  D, sp+4
0xfff400dc:  a8 80 04 ff  opcode for mov A, func1::func1_static
0xfff400e0:  00 00 00 00  immediate value for preceding mov
0xfff400e4:  a8 10 05 04  opcode for ld  D, A
0xfff400e8:  a8 91 05 01  opcode for add D, 1
0xfff400ec:  a8 11 05 04  opcode for st  D, A
0xfff400f0:  a8 9b 00 04  opcode for sub sp, 4
0xfff400f4:  a8 40 01 00  opcode for ret

(Start of file1.o's .rodata segment)
0xfff400f8:  48 6f 77 20  'H' 'o' 'w' ' '  variable lit001
0xfff400fc:  61 72 65 20  'a' 'r' 'e' ' '
0xfff40100:  79 6f 75 3f  'y' 'o' 'u' '?'
0xfff40104:  00 00 00 00  (null terminator) (3 more 0 bytes for 4-byte alignment)

(Start of file1.o's .romdata  segment)
0xfff40108:  00 00 00 03  variable init1
0xfff4010c:  00 00 00 04  variable init2
0xfff40110:  48 65 6c 6c  'H' 'e' 'l' 'l'  variable my_vstring
0xfff40114:  6f 2c 20 77  'o' ',' ' ' 'w'
0xfff40118:  6f 72 6c 64  'o' 'r' 'l' 'd'
0xfff4011c:  21 00 00 00  '!'  (null terminator) (two more 0 bytes for 4-byte alignment)
0xfff40120:  00 00 00 00  my_ptr.  After link, value will be address of lit001.
(Start of file2.o's .romdata segment)
0xfff40124:  00 00 00 11  variable init3
0xfff40128:  46 69 6e 65  'F' 'i' 'n' 'e'; variable other_string
0xfff4012c:  2c 20 74 68  ',' ' ' 't' 'h'
0xfff40130:  61 6e 6b 73  'a' 'n' 'k' 's'
0xfff40134:  2e 00 00 00  '.' (null terminator) (two more bytes for four-byte alignment)
0xfff40138:  00 00 00 00  variable func1_static, private to func1

(Start of file1.o's .data segment)
0x10040000:  00 00 00 03  variable init1
0x10040004:  00 00 00 04  variable init2
0x10040008:  48 65 6c 6c  'H' 'e' 'l' 'l'  variable my_vstring
0x1004000c:  6f 2c 20 77  'o' ',' ' ' 'w'
0x10040010:  6f 72 6c 64  'o' 'r' 'l' 'd'
0x10040014:  21 00 00 00  '!'  (null terminator) (two more 0 bytes for 4-byte alignment)
0x10040018:  00 00 00 00  my_ptr.  After link, value will be address of lit001.
(Start of file2.o's .data segment)
0x1004001c:  00 00 00 11  variable init3
0x10040020:  46 69 6e 65  'F' 'i' 'n' 'e'; variable other_string
0x10040024:  2c 20 74 68  ',' ' ' 't' 'h'
0x10040028:  61 6e 6b 73  'a' 'n' 'k' 's'
0x1004002c:  2e 00 00 00  '.' (null terminator) (two more bytes for four-byte alignment)
0x10040030:  00 00 00 00  variable func1_static, private to func1

(Start of file1.o's .bss segment)
0x10040034:  00 00 00 00  variable uninit1
0x10040038:  00 00 00 00  variable uninit2
0x1004003c:  00 00 00 00  First 4 bytes of my_zstring
...          ...          248 more bytes of my_zstring
0x10040138:  00 00 00 00  Last 4 bytes of my_zstring
(Start of file2.o's .bss segment)
0x1004013c:  00 00 00 00  variable uninit3
```

6.2. Re-writing symbol providers in the symbol tables

Now that we've laid out all the segments, we can renumber the first columns in the symbol tables:

```
0x10040000:  provide init1
0x10040004:  provide init2
0x10040008:  provide my_vstring
0x10040018:  provide my_ptr
0x10040018:  require lit001
0x10040034:  provide uninit1
0x10040038:  provide uninit2
0x1004003c:  provide my_zstring

0xfff400f8:  provide lit001
0xfff40080:  provide main
0xfff4009c:  provide L01
0xfff400ac:  require func1
0xfff400b0:  require L01

0x1004001c:  provide init3
0x10040020:  provide other_string
0x10040030:  provide func1::func1_static
0x1004013c:  provide uninit3
0xfff400bc:  provide func1
0xfff400d0:  require func1::func1_static
0xfff400e0:  require func1::func1_static

0xfff40000:  provide _start
0xfff40004:  require romdata_start
0xfff4000c:  require data_start
0xfff40014:  require romdata_end
0xfff4001c:  require romdata_start
0xfff40020:  provide data_copy
0xfff40024:  require data_copy_done
0xfff4003c:  require data_copy
0xfff40040:  provide data_copy_done
0xfff40044:  require bss_start
0xfff4004c:  require bss_end
0xfff40054:  require bss_start
0xfff40058:  provide bss_fill
0xfff4005c:  require bss_fill_done
0xfff4006c:  require bss_fill
0xfff40070:  provide bss_file_done
0xfff40074:  require stack_start
0xfff40078:  require main
0xfff4007c:  provide spin
0xfff4007c:  require spin
```

6.3. Resolving symbol requirers

Now that the segments are all laid out and the providers updated, we can make another pass resolving all the requirers. For example, 0xfff40074 requires the symbol `main`. So, we loop through the symbol table looking a provider of `main`. If there isn't one, the link fails with `undefined symbol`. If there is more than one, the link fails with `multiply defined symbol`.

Specifically, we now can make the following changes. 32-bit unresolved values get replaced by symbol-table entries – e.g. 0xfff40004 requires `romdata_start`, which is 0xfff40108, so those 32 bits of 0x00000000 get replaced by 0xfff40108. By contrast (for this fictitious processor), branch statements have their lower 16 bits unresolved, which is a count of 32-bit words from source to destination. For example, `main` is provided by `file1.o` at 0xfff40080, so we overwrite the lower 16 bits at 0xfff40078 and replace them with (0xfff40080 - 0xfff40078) / 4, which is 2.

```
...
0xfff40004:  ff f4 01 08  immediate value is now romdata_start
...
0xfff4000c:  10 04 00 00  immediate value is now data_start
...
0xfff40014:  ff f4 01 3c  immediate value is now romdata_end
...
0xfff4001c:  ff f4 01 08  immediate value is now romdata_start
...
0xfff40024:  28 30 00 07  opcode for bge data_copy_done
...
0xfff4003c:  a8 30 ff f9  opcode for b data_copy
...
0xfff40044:  10 04 00 34  immediate value is now bss_start
...
0xfff4004c:  10 04 01 40  immediate value is now bss_end
...
0xfff40054:  10 04 00 34  immediate value is now bss_start
...
0xfff4005c:  28 30 00 05  opcode for bge bss_fill_done
...
0xfff4006c:  a8 30 ff fb  opcode for b bss_fill
...
0xfff40074:  10 0f ff f0  immediate value is now stack_start
0xfff40078:  a8 31 00 02  opcode for blr  main
0xfff4007c:  a8 30 00 00  opcode for b spin; spin label
...
0xfff400ac:  a8 31 00 04  opcode for bl func1
0xfff400b0:  a8 30 ff fb  opcode for b  L01
...
0xfff400bc:  a8 9a 00 04  opcode for add sp, 4; start of func1
...
0xfff400d0:  10 04 00 30  immediate value is now func1::func1_static
...
0xfff400e0:  10 04 00 30  immediate value is now func1::func1_static
...
0xfff40120:  ff f4 00 f8  my_ptr.  After link, value is now the address of lit001.
...
0x10040018:  ff f4 00 f8  my_ptr.  After link, value is now the address of lit001.
```

7. Writing the linker output map file

The linker input map file contained some constraints on placement, but it didn't spell out every last detail. For example, although it said that the data segment should start at 0x10040000, it just said that the `.bss` segment should go after that. And it said nothing about, for example, the location of the `init2` variable.

When you're debugging embedded systems, you will often need to know what actually went where. This information is essential, for example, when you see an address in a logic-analyzer trace – and need to know what that is the address *of*.

We do this by taking the symbol tables from above, removing the requires and preserving the provides, and sorting them numerically. This gives:

```
Symbol table for myprog
Generated by FumblyFingers v 1.00
Date:  February 31, 1978
Time:  00:31:04

0x10040000:  data_start
       0x10040000:  init1
       0x10040004:  init2
       0x10040008:  my_vstring
       0x10040018:  my_ptr
       0x1004001c:  init3
       0x10040020:  other_string
       0x10040030:  func1::func1_static
0x10040034:  data_end

0x10040034:  bss_start
       0x10040034:  uninit1
       0x10040038:  uninit2
       0x1004003c:  my_zstring
       0x1004013c:  uninit3
0x10040140:  bss_end

0xfff40000:  text_start
       0xfff40000:  _start
       0xfff40020:  data_copy
       0xfff40040:  data_copy_done
       0xfff40058:  bss_fill
       0xfff40070:  bss_file_done
       0xfff4007c:  spin
       0xfff40080:  main
       0xfff4009c:  L01
       0xfff400bc:  func1
0xfff400f8:  text_end

0xfff400f8:  rodata_start
       0xfff400f8:  lit001
0xfff40108:  rodata_end

0xfff40108:  romdata_start
0xfff4013c:  romdata_end
```

Some linkers will generate an output-map file automatically; other must be requested to do so. Please consult your toolset's manual to find out how to get the linker to generate an output-map file – it will often be necessary while troubleshooting.

8. Writing the plain-binary file

At this point, the linker can write out a *plain binary file*. This is just a bit-for-bit copy of the `.text` segment starting at (board address) 0xfff40000, running through to the end of the `.romdata` segment at 0xfff4013c.

We don't need to include the data segment in the binary file: its contents are in the `.romdata` segment, which is in the file, and the instructions to copy `.romdata` to data are present in the `.text` segment (in `_start`). Likewise, we don't need to copy the `.bss` segment to the binary file: the instructions to zero-fill it are contained within `_start`, in the `.text` segment. Lastly, we don't need to copy the `.stack` segment: `_start` will set the stack pointer, and the other routines will push and pop the stack at runtime.

If we hex-dump the plain-binary file (see section **Error! Reference source not found.**, page **Error! Bookmark not defined.**, for a tool to do this) we'll see something like this:

```
00000000: a8 80 05 ff  ff f4 01 04  a8 80 06 ff  10 04 00 00 |................|
00000010: a8 80 04 ff  ff f4 01 38  a8 82 04 ff  ff f4 01 04 |.......8........|
00000020: a8 30 04 00  28 30 00 06  a8 10 05 07  a8 11 07 06 |.0..(0..........|
00000030: a8 91 05 04  a8 91 06 04  a8 82 04 04  a8 30 ff fa |.............0..|
00000040: a8 80 04 ff  10 04 00 34  a8 80 07 ff  10 04 01 40 |.......4.......@|
00000050: a8 82 07 ff  10 04 00 34  a8 30 07 00  28 30 00 05 |.......4.0..(0..|
00000060: a8 51 04 00  a8 92 07 04  a8 91 04 04  a8 30 ff fb |.Q...........0..|
00000070: a8 80 02 ff  10 0f ff f0  a8 31 00 02  a8 30 00 00 |.........1...0..|
00000080: a8 9a 00 10  a9 80 04 0c  a8 90 05 03  a8 11 05 04 |................|
00000090: a9 80 04 10  a8 90 05 04  a8 11 05 04  a9 10 05 0c |................|
000000a0: a9 11 05 04  a9 10 05 10  a9 11 05 08  a8 31 00 04 |.............1..|
000000b0: a8 30 ff fb  a8 9b 00 10  a8 40 01 00  a8 9a 00 04 |.0.......@......|
000000c0: a8 80 05 ff  de 00 00 ac  a9 11 05 04  a8 80 04 ff |................|
000000d0: 10 04 00 30  a8 10 05 04  a9 11 05 04  a8 80 04 ff |...0............|
000000e0: 10 04 00 30  a8 10 05 04  a8 91 05 01  a8 11 05 04 |...0............|
000000f0: a8 9b 00 04  a8 40 01 00  48 6f 77 20  61 72 65 20 |.....@..How are |
00000100: 79 6f 75 3f  00 00 00 00  00 00 00 03  00 00 00 04 |you?............|
00000110: 48 65 6c 6c  6f 2c 20 77  6f 72 6c 64  21 00 00 00 |Hello, world!...|
00000120: ff f4 00 f4  00 00 00 11  46 69 6e 65  2c 20 74 68 |........Fine, th|
00000130: 61 6e 6b 73  2e 00 00 00  00 00 00 00              |anks........|
```

We'll see in section **Error! Reference source not found.**, page **Error! Bookmark not defined.**, how this plain binary file gets loaded and executed at runtime.

Some key points to note:

- This is all the hardware knows about. It bears little resemblance to the original source code on page **Error! Bookmark not defined.**. All of our compiler, assembler and linker technology does no more and no less than to translate C and assembly code into something like this.

- It is likely to be unintelligible by itself, unless perhaps you know your processor's opcodes by heart. The linker-output map file and the hex dump can be invaluable in finding out what is where.

  For example, using the linker output map file, you can look at file offset 0x00000110 (file offset of `.romdata` copy of `.data` address 0x10040004) and realize, "There's the line `static int init2 = 4` from `file1.c`."

- Note that the left-hand column of the hex dump shows offsets from the start of the file – beginning at 0x000000. In one's head, one must add in the processor's memory offset (in this example, 0xfff40000).

- A hex dump shows you what the data looks like in hex and ASCII, but sometimes you want to have the instructions in the `.text` segment disassembled for you (see the next section for an example). You can use your cross-tools' disassembler tool (e.g. GNU's `objdump -d`). These three are the pillars for debugging in the logic analyzer (and elsewhere): The linker output map file, the hex dump, and the disassembly file.

Supposing your linker creates an ELF file `myprog.elf` and a plain binary `myprog.bin`, you can type something like the following (or use a shell script/batch file to create them):

```
PC prompt> cross-nm myprog.elf  | sort > myprog.map   map file
PC prompt> cross-objdump –d myprog.elf > myprog.dis   disassembly file
PC prompt> hex myprog.bin              > myprog.hex   hex dump
```

If your cross tools support a post-link command, you could put such a script there. Then, you'll always have these three useful files, and they'll always be current with the executable file.

9. Disassembly

Using your cross-tools' disassembler (e.g. GNU's `objdump -d`) you can get binary instructions turned back into something that looks like the original assembly source (whether hand-written by you, or generated by the compiler). If you disassemble the plain binary, or if you disassemble the ELF file and it's *stripped* (lacks debug symbols), then you'll get numerical addresses. For example:

```
00000000:  a8 80 05 ff  mov   X,  0xfff40108
00000004:  ff f4 01 08
00000008:  a8 80 06 ff  mov   Y,  0x10040000
0000000c:  10 04 00 00
00000010:  a8 80 04 ff  mov   A,  0xfff4013c
00000014:  ff f4 01 3c
00000018:  a8 82 04 ff  sub   A,  0xfff40108
0000001c:  ff f4 01 08
...
```

If you disassemble an ELF file that has symbols, you'll get a more informative dump. For example:

```
fff40000 <_start>:
       fff40000:  a8 80 05 ff  mov   X,  0xfff40108  <romdata_start>
       fff40004:  ff f4 01 08
       fff40008:  a8 80 06 ff  mov   Y,  0x10040000  <data_start>
       fff4000c:  10 04 00 00
       fff40010:  a8 80 04 ff  mov   A,  0xfff4013c  <romdata_end>
       fff40014:  ff f4 01 3c
       fff40018:  a8 82 04 ff  sub   A,  0xfff40108  <romdata_start>
       fff4001c:  ff f4 01 08
       ...
```

10. Intermediate files

What we call the "compiler" is really at least four separate programs:

- The *preprocessor*, which handles `#include`, `#define`, etc.
- The *compiler* per se, which reads preprocessed C source and generates assembly.
- The *assembler*, which turns assembly into object files.
- The *linker*, which links object files and libraries into an executable file.

Often, the only files you see on your disk after your build is done is the final executable file, and maybe the object files – since the subprograms clean up after themselves by default. Sometimes, though, for debugging, you want to see some of the intermediate files. The details vary from one compiler to another, but most compilers I've seen support the following options:

- `cc –E file.c > file.i`: Stop after preprocessing; output in `file.i`.
- `cc –S file.c`: Stop after compiling; assembly is in `file.s`.
- `cc –o file.c`: Stop after assembling; object file is `file.o`.

Also, using the disassembler as described above, you can map an object file or an executable file back into the assembly language. Either way, you can see the assembly statements generated by the compiler.

11. Writing an ELF file

There are several common output-file formats, two of which are plain binary and ELF. (I won't discuss S-record files in this document.) An ELF file contains all the segments that the plain binary has, preceded by a header that specifies where the segments start within the file, how big they are, and where they should be copied to at run time. Also, they may have debug symbols present. (Debug symbols are the kinds of thing that allow a debugger to encounter an address and know what it is the address of – e.g. variable name, source file and line number.)

ELF files are not directly executable. The first four bytes (the "magic number" for ELF files) are always 0x7f 0x45 0x4c 0x46 (DEL, then ASCII "E", "L","F") which is not likely to be a valid opcode for a processor. An ELF file must be executed by another program – already running – which knows what to do with it. (See, for example, `X:\lib\elf.c` and `execute_jump()` in `X:\avmon\cmd\cmdcore.c` for a very minimal ELF loader.) Specifically:

Your processor-reset code (Avmon for now, boot manager later) must be a plain binary.

Plusses:

- ELF files allow discontiguous `.text`, `.data` and `.bss` segments, if you should want that.
- If the ELF file contains symbolic information, a disassembler (e.g. GNU's `objdump -d`) can provide an informative disassembly, with function and variable names alongside numerical addresses.

Minuses:

- The ELF header is usually 64KB; also, debug symbols (if present) take up space. This makes it take more time to transfer to the board, and takes up more space in flash.
- ELF files aren't directly executable, so the first code that runs on the processor can't be an ELF file.

Personally, I prefer to have the linker output both types, in order to be able to do the steps described in section 8 on page 17.

12. Why?

By the end of this section on linking and loading, you may be asking, Why did we go through this? Do I really need to know this? The answer is an emphatic *yes*. When you are validating a board, anything can and does go wrong. Your perfectly bug-free code might be hanging up somewhere, through no fault of its own, or you might have a bug in your code, or both – or, most likely, you might not know *where* the problem is. When you are troubleshooting memory devices, and/or using a logic analyzer, you have the source code in a window on your PC, but you have just a bunch of bits running on the hardware. You will need to answer these two questions to make sense of what you're looking at:

- Where is my code?
- What are these bits?

12.1. Where is my code?

The information in this section enables you map the source code you see on your screen to the bits you see on the board. Knowing about segments enables you to know what parts of your code go into flash and what parts go into RAM. The linker output map and the plain binary files are indispensable in finding out what your code turned into.

12.2. What are these bits?

The reverse question also arises: How to take the bits you see as you troubleshoot and map them back to the source code. For example, your logic analyer shows you a sequence of addresses on the address bus, then the processor seems to halt. Where is it? What code is it executing? Here you can use the linker output map file and the plain binary file to find out what addresses belong to what code. You can use the disassembly of your plain binary (or the assembly listings from the compiler, as described above) to map opcodes back to assembly, back to C.