

Document Scope:

This document is an outline of how to produce absolute linked code / data that may be placed into Flash / PROM.

Comments:

I will take an example from what I have done for a bootloader. This is on the 68EZ328 platform, and you must be able to read 68K assembler to understand what is happening in the boot.S file.

Building absolute, ROM-able, code is an advanced topic and you can easily make mistakes. Probably the best way to build such code is to "breakpoint" the code, either with a debugger (ICE), or by printing critical address values out a serial port to a console. On my DragonBall design, I will print a signaling character out the serial port during the boot process, then place the processor in a forever loop (die: bra die). Once I get the indication that I have encountered my "breakpoint", I then invoke the BDM mode on the board and examine the memory contents using a kermi script.

How it is done:

First of all, my system memory map is this:

-----	- 0xffffffff
UNUSED	
-----	- 0x10007fff
32K EPROM	
-----	- 0x10000000
UNUSED	
-----	- 0x0203ffff
COMPACT FLASH	
-----	- 0x0201ffff
ETHERNET	
-----	- 0x02000000
UNUSED	
UNUSED	
-----	- 0x007fffff
8M DRAM	
-----	- 0x00000000

In order to build a boot loader, I needed to understand how my processor started up. The MC68EZ328 starts its "life" (out of reset) with the CSA0 line selecting ALL memory, anyplace you access will cause CSA0 to activate, so, I tied the 32K boot EPROM to CSA0. The first instruction fetched by the cpu is that of the first one at location 0x0000 of the boot EPROM.. Now I have control over the cpu, it is executing my opcodes.

What you have to realize is that the cpu does NOT differentiate between a branch relative or a jump absolute instruction at this point, it doesn't care, it will access ALL memory at ANY address location! The problem that you will first encounter is when you condition CSA0 to become the chip select for the 32k region of 0x10000000..0x1001FFFF, NOW, the code cannot randomly use jump address targets outside of that range without the danger of a bus fault.

Looking at the loader code, let's examine the criteria for allowing the boot code to run:

1. .text section must be located (linked) at a physical address, that is within the PROM, so that we can make calls to functions that we have written and the code we "land" on is the expected code.
2. .data section must be located within volatile storage (RAM) so that we can operate upon it, and we have to initialize the contents of that RAM so stuff like: 'char it[] = "here I am";' will work.
3. .bss section must also be located within volatile RAM so it can be modified, but unlike the .data section, we must ZERO the contents of the .bss section (this is expected of us). bss variables are those which have no initialization: 'char it [80];'.
4. .const section (IF ANY) we don't need to worry about as it should be placed within the .text section by default, if we don't explicitly mention it in the .ld file.

Testing the platform

To make all this happen, we must determine HOW our physical memory will be laid out and to specify WHERE in that memory that these things will reside (sections). I have done both of these things in determining my memory map (previous), some of the choices were purely arbitrary, others met the needs of the devices I used on my board. We can cause the various pieces of hardware to appear within (at) the locations specified in the memory map by initializing the various Chip Select registers. This you probably know already, if not, then read your docs on the cpu that you are using. What I have

Absolute linking with gcc & friends

found to be helpful is to do something like this to wiggle the chip selects and verify their activity with a scope or logic probe:

```
    move.l #0x2000000, %d0 ; Point to the Ethernet chip.
    move.l %d0, %a0
wiggle:
    move.b (%a0), %d0      ; Access CSB0 (pin 55).
    bra    wiggle         ; DO NOT JUMP! Keep it relative!
```

This will cause activity on that pin. This may seem silly to you, verifying that the chip selects are properly configured, but -- **NEVER ASSUME** --, always test, especially with a new board you are not familiar with! The goal here is to have a *high degree* of confidence in your hardware initialization *before* you attempt to do absolute linking.

BTW. If you haven't written your own crt0.S file (my case: boot.S), then you will be in big trouble attempting to construct a bootloader! Boot code has a *radically* different startup sequence than that of the stock crt0.o files you write applications against. The boot.S file can simply be something like:

```
#define ASSEMBLY
    .text
    bra    main
exit:
    bra    exit
```

Again, **do not** use hard addresses within the simple boot.S, you won't have the chip selects functional (other than default reset values), so take advantage of the *relative* branch instruction to get you into main() where you can initialize the chip selects by using C.

I would only advise **expert** C coders to build their initialization code in C!!! To ensure that any instructions that gcc emits are *relative* branches, prior to the initialization of the chip selects, you would have to dump the gcc output into an assembly source file, then read the source and look for gotcha's like an absolute jump instruction.

A word about initializing chip selects, always initialize your ROM (Flash, EPROM, etc) that contains your boot code first, that way you don't activate, say DRAM, and collide with CSA0 that is still configured to activate everywhere!

Making the .ld file

Once we are confident that the hardware is functional, we can now tackle how to setup our linker file to set where stuff is going to be placed (at **what** address). The linker .ld file is only for adjustment of the address references within the .text code so that it properly knows where (what address) functions and data are to be found. The .ld file **does not** cause the data to be moved to those locations, only describes where things may be found (subroutines, variables, free memory...).

For example, my bootstrap.ld:

```
MEMORY {
    ram      : ORIGIN = 0x00200000, LENGTH = 0x0001FFFF
    flash    : ORIGIN = 0x10000000, LENGTH = 0x007FFFFFFF
    dram     : ORIGIN = 0x0021FFFF, LENGTH = 0x000E0000
    eram     : ORIGIN = 0x007FFFF0, LENGTH = 1
}

SECTIONS {
    .text : {
        _stext = . ;
        *(.text)
        _etext = . ;
        __data_rom_start = ALIGN(4);
    } > flash

    .data : {
        _sdata = . ;
        *(.data)
        _edata = . ;
    } > ram

    .bss : {
        _sbss = . ;
        *(.bss)
        *(COMMON)
        _ebss = . ;
    } > ram

    .eram : {
        _eramend = . ;
    } > eram
}
```

Before I dissect this critical file, you have to understand the strategy of my boot process:

1. The first 1 Megabyte of data (linux kernel) is pulled out of the Compact Flash and loaded into DRAM starting with address 0x0.
2. Then the kernel is jumped to at location 0x400, the code at location 0x400 in the kernel is the beginning of my crt0_cf.S file from arch/m68knommu/platform/68EZ328/ez328lcd/.

Knowing that the first 1+ Meg of DRAM is going to be used to hold the data loaded from the CF drive, I start my boot.S RAM usage for the .data + .bss sections at 0x200000. This should give me plenty of room to let future kernels "grow". The .text section contains any .const data + bootloader code, so that can stay in ROM at location 0x1000000 (this is where the CSA0 was finally set for).

These two key sections of the linker .ld file, **MEMORY** + **SECTIONS**, are exactly what they infer: The **MEMORY** dictates a memory mapping of regions of the machine and give *symbolic* names. The **SECTIONS** area dictates where the *sections* (.text, .const, .data & .bss) are to be placed, and in what order within each **SECTION**.

FLASH: .text section is all sections (named by gcc) to have *(.text) , or *anything*.text in its section name, to be placed within the **MEMORY** location I arbitrarily named this area *flash*. The flash **SECTION** is **MEMORY** mapped into 0x1000000..0x1007fff. NOTE that I also have placed some symbols, _stext + _etext, within the flash **SECTION**? Those symbols can also be resolved by the linker and I can (and do) refer to some of those symbols in my boot.S code.

RAM: is the next two **SECTIONS**, the same situation as explained in the CODE above. NOTE that the ram **MEMORY** area is first filled with the variables from the .data section then it is "filled" with the .bss section.

ERAM: this is to set the symbol _ramend so that I can refer to it in boot.S to set the processor stack to the top of DRAM.

Building the binary

Now we need to create a ROM-able image that we can program into Flash / EPROM, and later copy the RAM section into DRAM and init the .bss section to ZEROs. To do this, you must examine my Makefile, the makefile contains a useage of the objcopy utility in such a way as to strip out the desired sections and place them into separate files:

```
boot.bin: bootstrap.coff
$(OBJCOPY) -O binary --remove-section=.data --remove-section=.bss \
    bootstrap.coff boot.text
$(OBJCOPY) -O binary --remove-section=.text --remove-section=.bss \
    bootstrap.coff boot.data
cat boot.text boot.data > boot.bin
```

First statement is to strip out the .data + .bss sections from the file bootstrap.coff and place the remaining sections (.text) into the boot.text file. The next statement is to strip out the .text + .bss sections and place the remaining sections into boot.data file. We don't care about .bss, it will be filled with zeros anyway (later we do that). Once we have these sections isolated, we cat the files together to form the binary image boot.bin.

... Now. You are probably wondering why we had to strip these out? Well, the coff file format is not usable as boot code, it contains a **lot**(!) of extra information and is not able to run as-is. We use objcopy to convert between the coff format and that of an executable, **but**, we cannot just do a:

```
m68k-coff-objcopy -O binary bootstrap.coff boot.bin
```

or, we end up with a 254 Megabyte file!! Why is it 254 Meg? That is the distance between the start of RAM (0x200000) to the end of the EPROM (approx. 0x10007fff), 0x200000..0x10000000 is a range of 254Meg! The objcopy does not care about the fact that the file is huge, it just ZERO fills the unused areas between the end of .data to the start of .text!

Putting memory in its' place

Moving on, now that we have our .text + .data sections compressed within a rom, it is time to go back to boot.S and see how it will all come together. The .text section we

Absolute linking with gcc & friends

don't have to worry about, it is already linked at location 0x10000000 and we will configure our chip select CSA0 to that location in memory. The .data & .bss sections are the problem, the linker told the .text code that this stuff would be at 0x200000 and we currently have it within our EPROM at somewhere above 0x10000000! How do we correct this? We simply copy the .data section down to where it should be within DRAM!

```
    moveal    #_sbss, %a0
    moveal    #_ebss, %a1
    /* Fill .bss with ZEROs until %a0 == %a1 */
L1:
    movel    #0, %a0@+
    cmpal    %a0, %a1
    bhi     L1
    /* Copy data segment from ROM to RAM */
    moveal    #__data_rom_start, %a0
    moveal    #_sdata, %a1
    moveal    #_edata, %a2
    /* Copy %a0 to %a1 until %a1 == %a2 */
L2:
    movel    %a0@+, %d0
    movel    %d0, %a1@+
    cmpal    %a1, %a2
    bhi     L2
lp:
```

Remember those symbols I mentioned before that I placed within the linker .ld file? Well here they come in handy, I can now point to where the .data section is within the ROM so I know where it is, ALSO, I can tell how big the .bss section is so I can ZERO it out! You must ZERO the bss section as the gcc compiler assumed that this will be done.

That's all: a mini-lesson on how to use the gcc + friends to perform absolute linking!