Memory Sections

Weak Bindings

FIXME: need to discuss the .weak directive.

The following describes the various sections available.

The .text Section

The .text section contains the actual machine instructions which make up your program. This section is further subdivided by the .initN and .finiN sections dicussed below.

Note:

The avr-size program (part of binutils), coming from a Unix background, doesn't account for the .data initialization space added to the .text section, so in order to know how much flash the final program will consume, one needs to add the values for both, .text and .data (but not .bss), while the amount of pre-allocated SRAM is the sum of .data and .bss.

The .data Section

This section contains static data which was defined in your code. Things like the following would end up in .data:

```
char err_str[] = "Your program has died a horrible death!";

struct point pt = { 1, 1 };
```

It is possible to tell the linker the SRAM address of the beginning of the .data section. This is accomplished by adding -Wl,-Tdata,*addr* to the avr-gcc command used to the link your program. Not that addr must be offset by adding 0x800000 the to real SRAM address so that the linker knows that the address is in the SRAM memory space. Thus, if you want the .data section to start at 0x1100, pass 0x801100 at the address to the linker. [offset explained]

Note:

When using malloc() in the application (which could even happen inside library calls), additional adjustments are required.

The .bss Section

Uninitialized global or static variables end up in the .bss section.

The .eeprom Section

This is where eeprom variables are stored.

The .noinit Section

This sections is a part of the .bss section. What makes the .noinit section special is that variables which are defined as such:

```
int foo __attribute__ ((section (".noinit")));
```

will not be initialized to zero during startup as would normal .bss data.

Only uninitialized variables can be placed in the .noinit section. Thus, the following code will cause avr-gcc to issue an error:

```
int bar __attribute__ ((section (".noinit"))) = 0xaa;
```

It is possible to tell the linker explicitly where to place the .noinit section by adding -Wl,--section-start=.noinit=0x802000 to the avr-gcc command line at the linking stage. For example, suppose you wish to place the .noinit section at SRAM address 0x2000:

```
 $ avr-gcc ... -Wl,--section-start=.noinit=0x802000 ...
```

Note:
: Because of the Harvard architecture of the AVR devices, you must manually add 0x800000 to the address you pass to the linker as the start of the section. Otherwise, the linker thinks you want to put the .noinit section into the .text section instead of .data/.bss and will complain.

Alternatively, you can write your own linker script to automate this. [FIXME: need an example or ref to dox for writing linker scripts.]

The .initN Sections

These sections are used to define the startup code from reset up through the start of main(). These all are subparts of the .text section.

The purpose of these sections is to allow for more specific placement of code within your program.

Note:
: Sometimes, it is convenient to think of the .initN and .finiN sections as functions, but in reality they are just symbolic names which tell the linker where to stick a chunk of code which is *not* a function. Notice that the examples for asm and C can not be called as functions and should not be jumped into.

The **.initN** sections are executed in order from 0 to 9.

.init0:
: Weakly bound to __init(). If user defines __init(), it will be jumped into immediately after a reset.

.init1:
: Unused. User definable.

.init2:
: In C programs, weakly bound to initialize the stack, and to clear __zero_reg__ (r1).

.init3:
: Unused. User definable.

.init4:

For devices with > 64 KB of ROM, .init4 defines the code which takes care of copying the contents of .data from the flash to SRAM. For all other devices, this code as well as the code to zero out the .bss section is loaded from libgcc.a.

.init5:
Unused. User definable.

.init6:
Unused for C programs, but used for constructors in C++ programs.

.init7:
Unused. User definable.

.init8:
Unused. User definable.

.init9:
Jumps into main().

The .finiN Sections

These sections are used to define the exit code executed after return from main() or a call to exit(). These all are subparts of the .text section.

The **.finiN** sections are executed in descending order from 9 to 0.

.finit9:
Unused. User definable. This is effectively where _exit() starts.

.fini8:
Unused. User definable.

.fini7:
Unused. User definable.

.fini6:
Unused for C programs, but used for destructors in C++ programs.

.fini5:
Unused. User definable.

.fini4:
Unused. User definable.

.fini3:
Unused. User definable.

.fini2:
Unused. User definable.

.fini1:
Unused. User definable.

.fini0:

Goes into an infinite loop after program termination and completion of any _exit() code (execution of code in the .fini9 -> .fini1 sections).

Using Sections in Assembler Code

Example:

```
#include <avr/io.h>

        .section .init1,"ax",@progbits
        ldi       r0, 0xff
        out       _SFR_IO_ADDR(PORTB), r0
        out       _SFR_IO_ADDR(DDRB), r0
```

Note:

The ,"ax",@progbits tells the assembler that the section is allocatable ("a"), executable ("x") and contains data ("@progbits"). For more detailed information on the .section directive, see the gas user manual.

Using Sections in C Code

Example:

```
#include <avr/io.h>

void my_init_portb (void) __attribute__ ((naked)) \
    __attribute__ ((section (".init3")));

void
my_init_portb (void)
{
        PORTB = 0xff;
        DDRB = 0xff;
}
```

Note:

Section .init3 is used in this example, as this ensures the inernal __zero_reg__ has already been set up. The code generated by the compiler might blindly rely on __zero_reg__ being really 0.