Program Development Tools

The GNU (GNU's Not Unix) Toolchain

The GNU toolchain has played a vital role in the development of the Linux kernel, BSD, and software for embedded systems.

The GNU project produced a set of programming tools.

Parts of the toolchain we will use are:

- *bash*: free Unix shell (Bourne-again shell). Default shell on GNU/Linux systems and Mac OSX. Also ported to Microsoft Windows.
- *make*: automation tool for compilation and build
- *gcc* (GNU Compiler Collection: suite of compilers for many programming languages, such as *avr-gcc*.
- *binutils*: Suite of tools including linker (*ld*), assembler (*gas*)
- *gdb*: Code debugging tool
- avr_libc: Subset of standard C library for Atmel AVR.

Program Development Tools

How an executable image is built

The process of converting source code to an executable binary image requires several steps, each with its own tool.

Compile:

C language source files are compiled by *avr-gcc* and assembled by *gas*. The assembler is usually called by *avr-gcc*.

Link:

The object files produced from the compile step must be linked together by *ld* to produce a single object file, called the *relocatable program*. The linker can optionally be called by *avr-gcc*.

Relocate:

Physical memory addresses are assigned to the relative offsets within the relocatable program. This is also handled by *ld*.

Program Development Tools

How a executable image is built – some details

avr-gcc is actually a cross compiler. (What is that?)

The output of the cross compiler is an object file. (*program.o*)

Object file:

- -a binary file that contains instructions and data from the language translation
- -cannot be executed directly
- -an incomplete image of the program (Why?)

The object file contains separate section containing code or data

- -all executable code is in the *.text* section
- -initialized global variables are in the *.data* section
- -uninitialized global variables are in the *.bss* section

Program Development Tools

How a executable image is built – some details

Often we have multiple ".c" files that result in multiple ".o" files.

Each ".c" file was compiled into a separate ".o" file.

Code in one file may call a function in code that is within another file. How can it know about the function? The compiler has to mark it to be resolved later.

All the object files must be combined to resolve the symbol names. The linker (*ld*) accomplishes this.

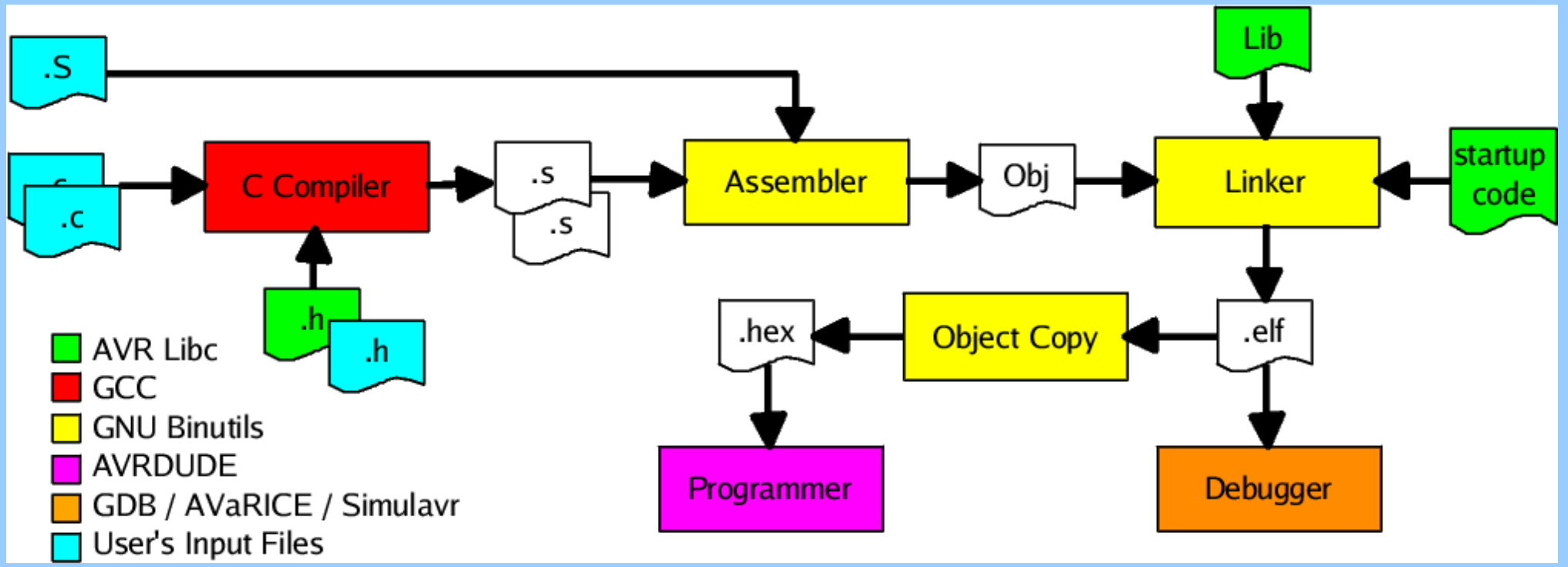
The output of the linker is a new object file (program.o) that contains all the code and data from all the (subprogram.o) files. It merges all the .text, .data, .bss sections.

This is usually in either:

-Common Object File Format (COFF) (*program.coff*) or
-Executable and Linkable Format (ELF) (*program.elf*)

Program Development Tools

How a executable image is built – some details



Program Development Tools

How a executable image is built – some details

One other job the linker does is to insert *startup code*.

Startup code is a small block of assembly code that prepares the way for the execution of software written in a high-level language.

For example, C programs expect a stack to exist. Space for the stack must be allocated before the C code can be run.

Startup code for C programs usually consist of the following series of actions:

- -disable interrupts (*after reset, AVR has interrupts off*)
- -copy initialized data from ROM to RAM (*Why do we have to do this?*)
- -zero out the uninitialized data area
- -allocate space for and initialize the stack
- -call `main();`

We can see these parts of the startup code if we compile in such a way to produce the .lst file

Program Development Tools

How a executable image is built – linker handiwork

From the first part of *lab1_code.lst*:

```
Disassembly of section .text:

00000000 <__vectors>:
   0: 0c 94 46 00    jmp   0x8c  ; 0x8c <__ctors_end>
   4: 0c 94 65 00    jmp   0xca  ; 0xca <__bad_interrupt>
   8: 0c 94 65 00    jmp   0xca  ; 0xca <__bad_interrupt>
   c: 0c 94 65 00    jmp   0xca  ; 0xca <__bad_interrupt>
  10: 0c 94 65 00    jmp   0xca  ; 0xca <__bad_interrupt>
  14: 0c 94 65 00    jmp   0xca  ; 0xca <__bad_interrupt>
  18: 0c 94 65 00    jmp   0xca  ; 0xca <__bad_interrupt>
```

And further down the page....

```
000000ca <__bad_interrupt>:
  ca: 0c 94 00 00    jmp   0  ; 0x0 <__heap_end>
```

The linker inserts *jump* instructions in unused interrupt slots of the interrupt vector table so if an interrupt occurs, it resets the processor.

Program Development Tools

How a executable image is built – linker handiwork

Further down in *lab1_code.lst*:

```
0000008c <__ctors_end>:
8c:  11 24  eor   r1, r1    #clear r1
8e:  1f be  out   0x3f, r1  #clear SREG, disable global ints
90:  cf ef  ldi   r28, 0xFF #
92:  d0 e1  ldi   r29, 0x10 # r29,r28 <= 0x10FF
94:  de bf  out   0x3e, r29 #
96:  cd bf  out   0x3d, r28 #stack pointer <= 0x10FF
#SRAM extends from 0x10FF down
```

(comments added after the fact)

The linker inserts code to initialize the stack pointer to the top of SRAM space, 0x10FF.

Program Development Tools

How a executable image is built – linker handiwork

Yet further down in *lab1_code.lst*:

```
00000098 <__do_copy_data>:
98:  11 e0  ldi  r17, 0x01 #load R17 with count of bytes to move
9a:  a0 e0  ldi  r26, 0x00 #load X register (R27,R26) with RAM address
9c:  b1 e0  ldi  r27, 0x01
9e:  e2 e5  ldi  r30, 0x52 #load Z register (R31,R30) FLASH address
a0:  f1 e0  ldi  r31, 0x01
a2:  00 e0  ldi  r16, 0x00
a4:  0b bf  out  0x3b, r16 #set access to lower 64K of pgm memory
a6:  02 c0  rjmp .+4    ;  0xac <__do_copy_data+0x14>
a8:  07 90  elpm r0, Z+    #load R0 with data pointed to by Z reg, post inc Z reg
aa:  0d 92  st   X+, r0    #location pointed to by X reg loaded from R0, inc X reg
ac:  a0 30  cpi  r26, 0x00 #compare R26 with 0x0
ae:  b1 07  cpc  r27, r17  #compare with carry r27 and r17
b0:  d9 f7  brne .-10   ;  0xa8 <__do_copy_data+0x10>  #branch if not equal (Zero flag
set)
```

The linker inserts code to create the initialized global data storage by copying data from flash to SRAM.

Program Development Tools

How a executable image is built – linker handiwork

Even further down in *lab1_code.lst*:

```
000000b2 <__do_clear_bss>:
 b2:  11 e0          ldi   r17, 0x01
  b4: a0 e0          ldi   r26, 0x00 #load X reg
  b6: b1 e0          ldi   r27, 0x01
  b8: 01 c0          rjmp  .+2   <.do_clear_bss_start>
000000ba <.do_clear_bss_loop>:
  ba: 1d 92          st X+, r1
000000bc <.do_clear_bss_start>:
  bc: a2 30          cpi   r26, 0x02
  be: b1 07          cpc   r27, r17
  c0: e1 f7          brne  .-8   <.do_clear_bss_loop>
  c2: 0e 94 81 00    call  0x102 <main>
  c6: 0c 94 a8 00    jmp   0x150 <_exit>
```

The linker inserts code to clear the uninitialized global variables section.