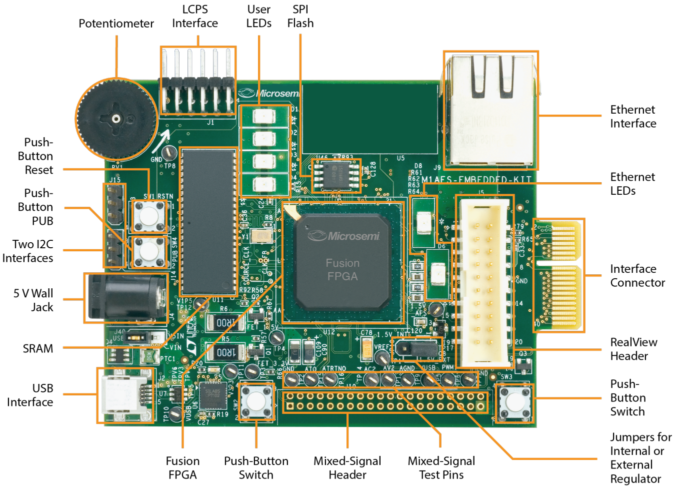


Embedded Software Fundamentals

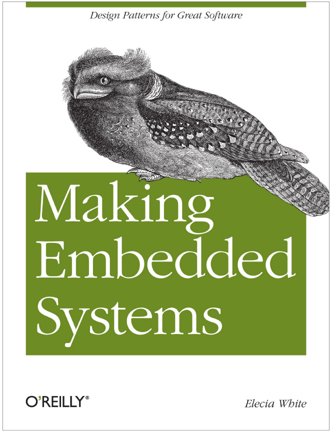
How does code get converted into ones and zeroes?

Kizito NKURIKIYEYEZU, Ph.D.

Reading material

Read the following resources available on the course platforms

1. Flowchart and software pseudocode which is uploaded on the course platform
2. Chapter 1 of White, E. (2011). Making Embedded Systems: Design Patterns for Great Software. " O'Reilly Media, Inc.".



Embedded software development

- Host Machine
- Development Environments
- Compiler Toolchain
- Debuggers
- Development Kits
- Version Control

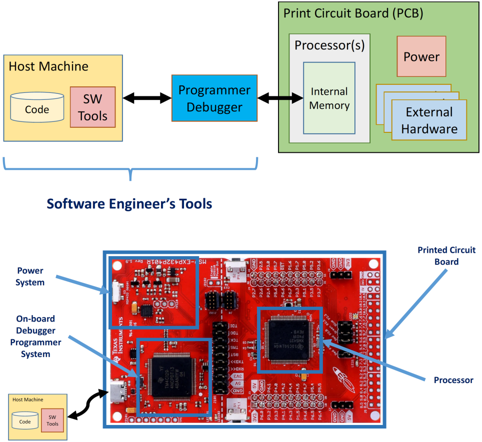


FIG 1. Components of an embedded development

Modules of a typical embedded software

- The software is organized in layers
- Each layer assumes specific functionality
- Modules are described in C-files (.c)
- Definitions are described in header files (.h)
- Functions interact with other modules
- Eventually interact with Hardware

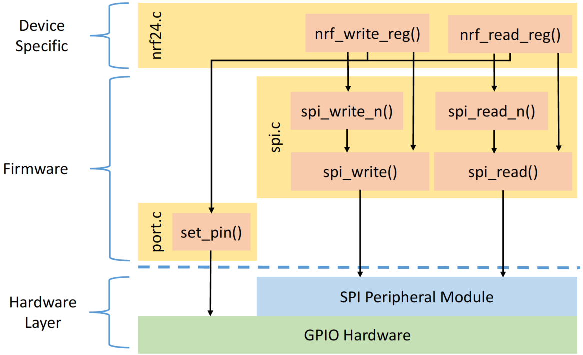
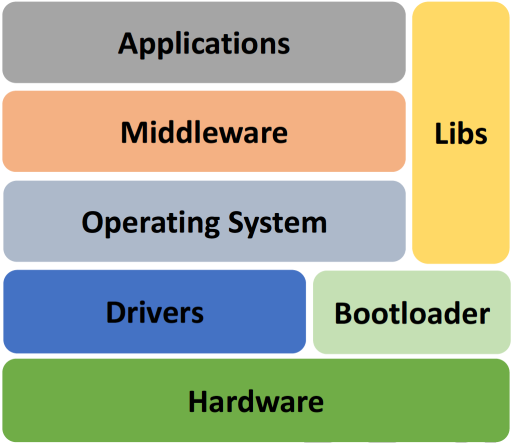


FIG 2. Layers of an embedded system software

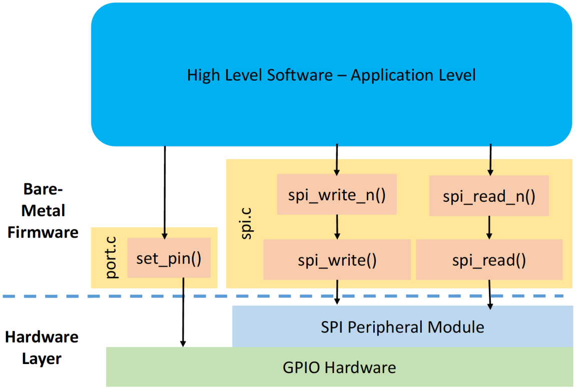
Embedded system software in layers

- Device Drivers
  - Interface to hardware layers
  - Hardware Abstraction Layer (HAL)
- Code Booting
- Real-time operation system (RTOS)
  - Abstracts High from Low levels
  - Scheduling
  - Resource management
- Libraries for shared code



Hardware Abstraction

- Low level and bare-Metal Firmware
- Hardware Abstraction Layer
- Platform Independence
- High quality and portable software
  - Maintainable
  - Testable
  - Portable
  - Robust
  - Efficient
  - Consistent



Embedded programming languages

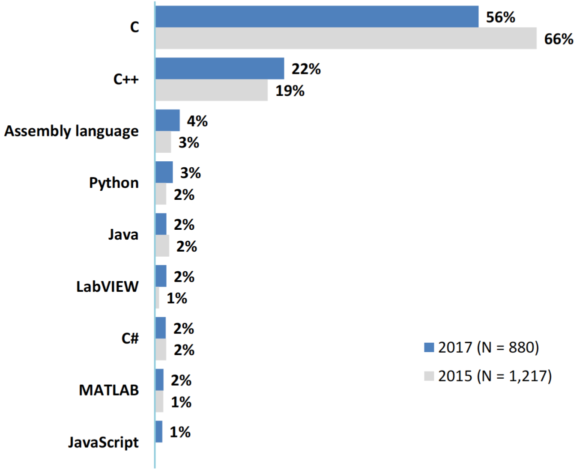


FIG 3. Top embedded programming languages

ASPENCORE. (2017). 2017 Embedded Markets Study Integrating IoT and Advanced Technology Designs, Application Development & Processing Environments. April, 1–102.

Why C?

- Availability of compilers for almost any MCU
- Small executable
- Deterministic resource use (e.g., no dynamic memory allocation)
- Efficient Memory Management
- Timing-centric operations
- Direction Hardware/IO Control
- Optimized execution
- **Note:** Modern C++ is as efficient as C and I believe it will slowly replace C in the future. For details see **Kormanyos, C. (2018). Real-time C++: efficient object-oriented and template microcontroller programming**

FIG 4. C can be used even on very small micro-controllers
The ATtiny20-UUR is an AVR micro-controller that is smaller than a grain of rice. It is an 8-Bit IC that runs at 12MHz 2KB (1K x 16) FLASH and 12-WLCSP (1.56x1.4)

Embedded software development process

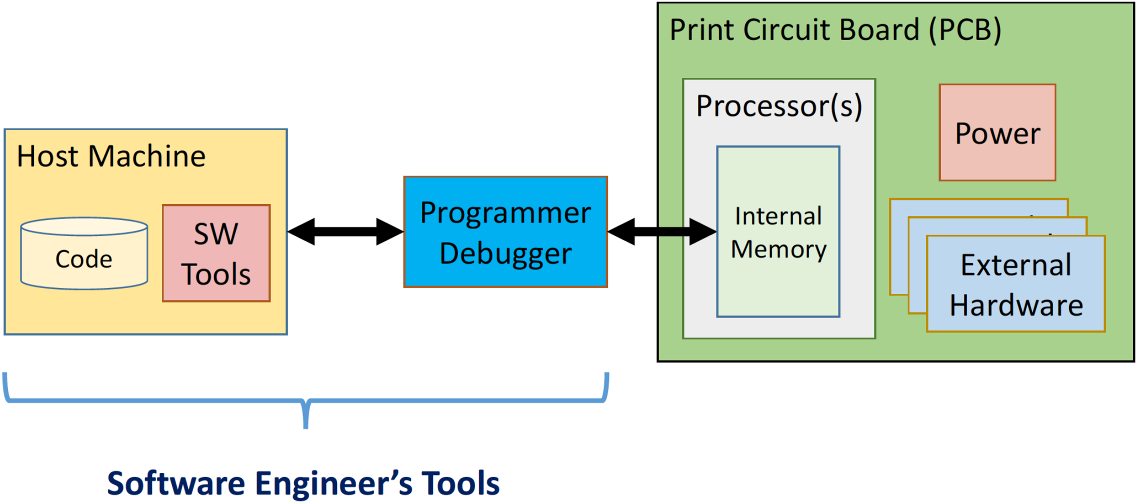


FIG 5. Embedded System Development Platform
The host machine contains the build environment for an embedded system. It contains a cross compiler and a cross debugger. The debug allows communication between the target processor through a special processor interface, the JTAG

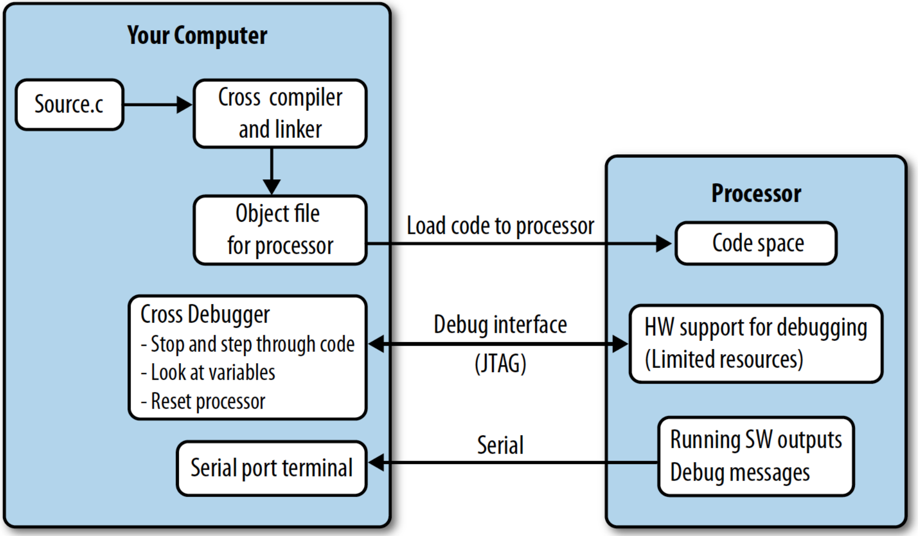


FIG 6. Computer and target processor

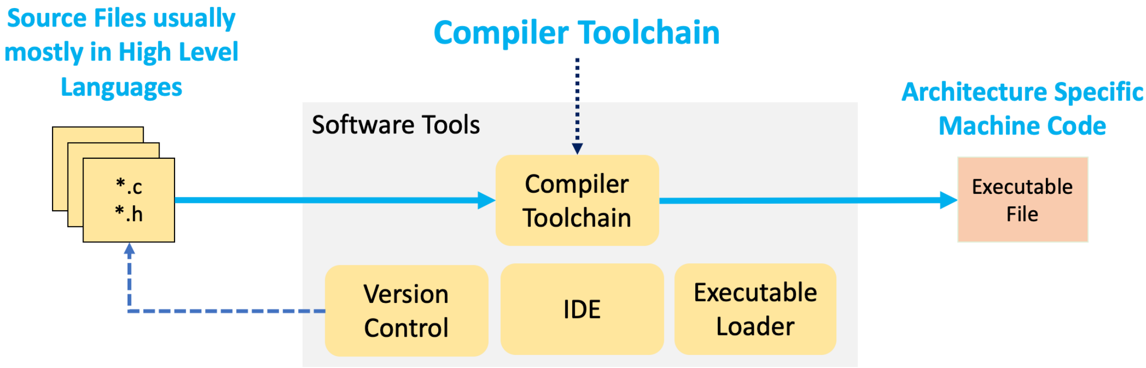


FIG 7. Software tools

The software tools include compiler toolchain (e.g., AVR GCC, gdb make files), linker, emulators, simulators, SDK, text editors/IDE, version control, etc

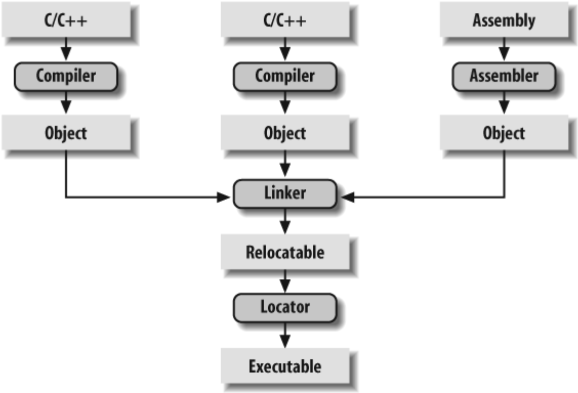


FIG 8. Detailed embedded C compilation process

The C preprocessor transform the program before actual compilation. The compiler translate the source code into opcode (object files) for the target processor. The linker combine these object files and resolve all of the unresolved symbols. The locator assign physical memory addresses to each of the code and data and produce an output file containing a binary memory image that can be loaded into the target ROM.

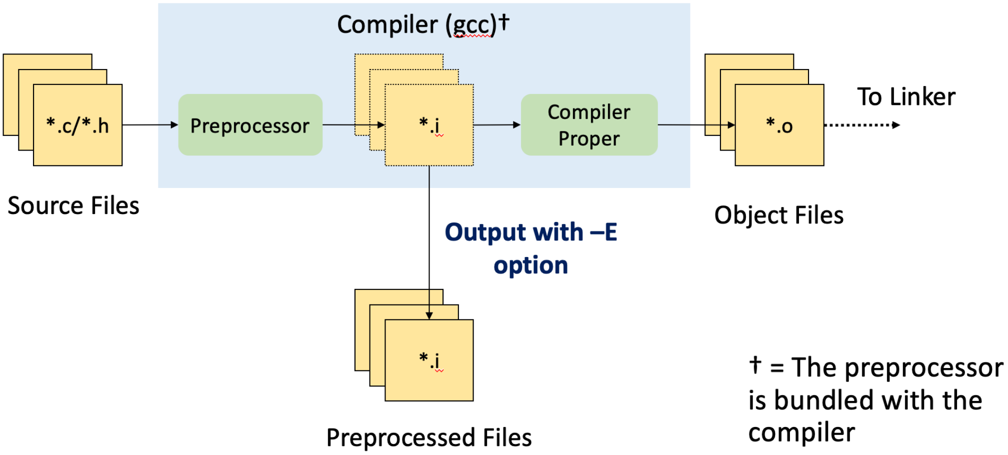


FIG 9. The role of a preprocessor

The C preprocessor is the macro preprocessor for the C compiler. The preprocessor provides the ability for the inclusion of header files, macro expansions, conditional compilation, and line control.

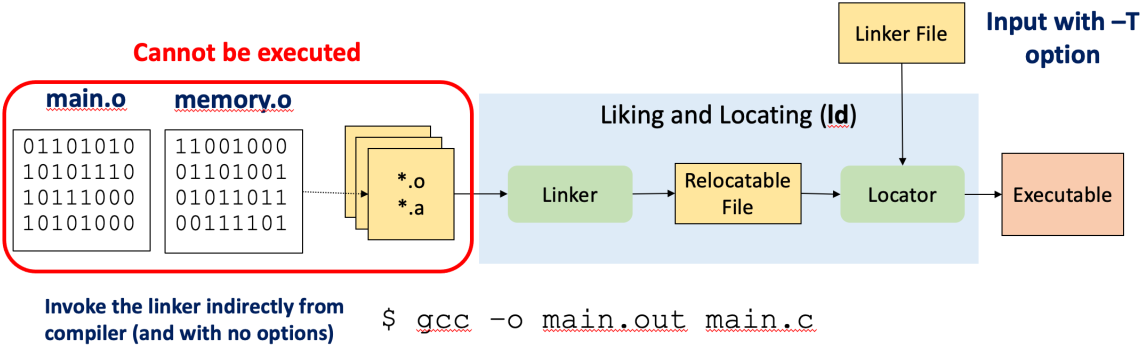


FIG 10. The role of a linker

The linker combines all of objects files into a single executable object code uses symbols to reference other functions/variables

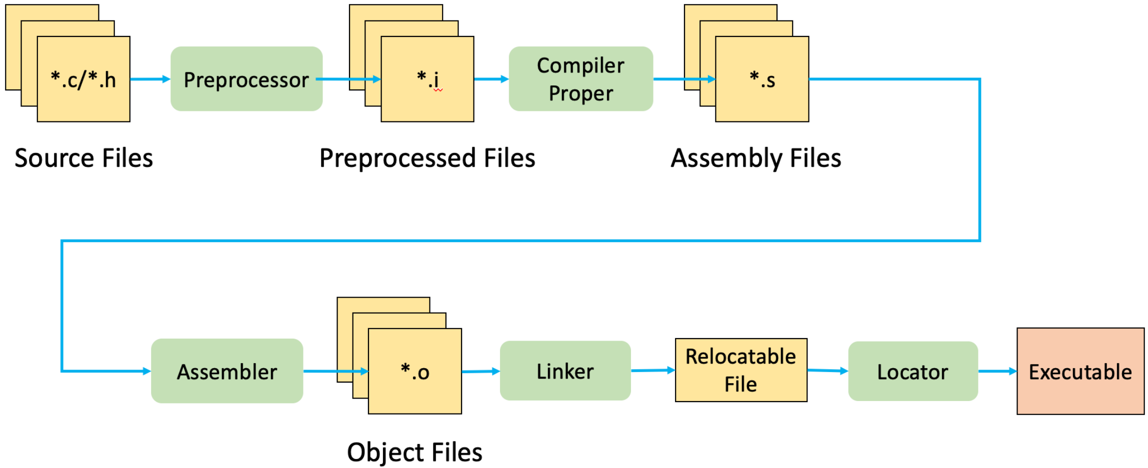


FIG 11. Linear detailed embedded C compilation process
The compiler translate the source code into opcode (object files) for the target processor. The linker combine these object files and resolve all of the unresolved symbols. The locator assign physical memory addresses to each of the code and data and produce an output file containing a binary memory image that can be loaded into the target ROM.

Code compilation using GNU Toolsets

- A computer only understand a set of instructions in a numeric format, typically called machine code

```c
#include <stdio.h>
int main() {
  printf("Hello, World!");
  return 0;
}
```

Listing 1. Source code

[1] https://www3.ntu.edu.sg/home/ehchua/programming/cpp/gcc_make.html

- A computer only understand a set of instructions in a numeric format, typically called machine code

```c
#include <stdio.h>
int main() {
  printf("Hello, World!");
  return 0;
}
```

Listing 2. Source code

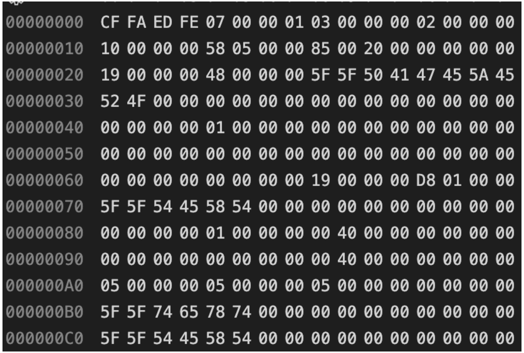


FIG 12. Machine code

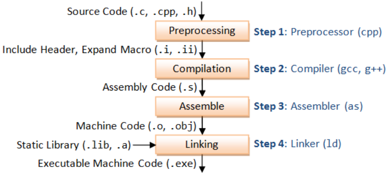
- The GCC compiler—The GNU Compiler Collection[1]—is often used for compilating embedded system

[1] https://www3.ntu.edu.sg/home/ehchua/programming/cpp/gcc_make.html

The preprocessor

- First stage of the compilation process
- Removes all the comments
- Include any #include files (typical the .h header file)
- Expands all the macros

```
gcc -E hello.c > hello.i
```



- Compilation—Compiles the pre-processed source code into assembly code for a specific processor

```
gcc -S hello.i
```

- Assembler converts the assembly code into machine code in the object file

```
as -o hello.o hello.s
```

- Linker links the object code with the library code to produce an executable file

```
    gcc -O hello.o
```

- Note: You can generate all intermediate files with the following command

```
    gcc -save-temps hello.c
```

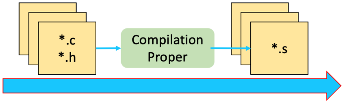
Introduction to Build Systems using AVR GNU Toolsets

Translation of C code into machine code

```
#include <avr/io.h>
int main (void){
    DDRB |= _BV(DDB0);
    while(1) {
        PORTB ^= _BV(PB0);
       _delay_ms(500);
    }
}
```

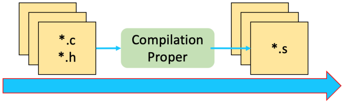
Translation of C code into machine code

```c
#include <avr/io.h>
int main (void){
    DDRB |= _BV(DDB0);
    while(1) {
        PORTB ^= _BV(PB0);
       _delay_ms(500);
    }
}
```



Translation of C code into machine code

```
#include <avr/io.h>
int main (void){
    DDRB |= _BV(DDB0);
    while(1) {
        PORTB ^= _BV(PB0);
       _delay_ms(500);
    }
}
```



:0C000000B89A91E088B38
:00000001FF

Translation of C code into machine code

GCC compiles a C/C++ program into executable in 4 steps:

1. **Pre-processing**—via the AVR GNU C Preprocessor (avr-cpp), which includes the headers (#include) and expands the macros (#define).

   avr-cpp -mmcu=attiny13 blink.c > blink.i

   The resultant intermediate file *blink.i* contains the expanded source code.

Translation of C code into machine code

GCC compiles a C/C++ program into executable in 4 steps:

1. **Pre-processing**—via the AVR GNU C Preprocessor (avr-cpp), which includes the headers (#include) and expands the macros (#define).

   avr-cpp -mmcu=attiny13 blink.c > blink.i

   The resultant intermediate file *blink.i* contains the expanded source code.

2. Compilation—the compiler compiles the pre-processed source code into assembly code for a specific processor.

   avr-gcc -S blink.i >blink.s

   The -S option specifies to produce assembly code, instead of object code. The resultant assembly file is "blink.s".

Translation of C code into machine code

GCC compiles a C/C++ program into executable in 4 steps:

1. **Pre-processing**—via the AVR GNU C Preprocessor (avr-cpp), which includes the headers (#include) and expands the macros (#define).

   avr-cpp -mmcu=attiny13 blink.c > blink.i

   The resultant intermediate file *blink.i* contains the expanded source code.

2. Compilation—the compiler compiles the pre-processed source code into assembly code for a specific processor.

   avr-gcc -S blink.i >blink.s

   The -S option specifies to produce assembly code, instead of object code. The resultant assembly file is "blink.s".

3. **Assembly** —the assembler (avr-as) converts the assembly code into machine code in the object file "hello.o".

   avr-as -o blink.o blink.s

Translation of C code into machine code

4. Linker: Finally, the linker links the object code with the library code to produce an executable and linkable format (.elf) file "blink.elf".

   avr-gcc blink.o -o blink.elf

   This generates an .elf file isn't directly executable by the MCU. Thus, one needs to extract the machine code from it in the Intel Hex format

   avr-objcopy -O ihex -R .eeprom blink.elf blink.ihex

Notes:

- You can see the detailed compilation process by enabling -v (verbose) option. For example,

  avr-gcc -v -mmcu=attiny13 -o blink.bin blink.c

Translation of C code into machine code

4. Linker: Finally, the linker links the object code with the library code to produce an executable and linkable format (.elf) file "blink.elf".

   avr-gcc blink.o -o blink.elf

   This generates an .elf file isn't directly executable by the MCU. Thus, one needs to extract the machine code from it in the Intel Hex format

   avr-objcopy -O ihex -R .eeprom blink.elf blink.ihex

Notes:

- You can see the detailed compilation process by enabling -v (verbose) option. For example,

  avr-gcc -v -mmcu=attiny13 -o blink.bin blink.c

- You can Generate all intermidiate files

  avr-gcc -mmcu=attiny13 -save-temps blink.c

Translation of C code into machine code

4. Linker: Finally, the linker links the object code with the library code to produce an executable and linkable format (.elf) file "blink.elf".

   avr-gcc blink.o -o blink.elf

   This generates an .elf file isn't directly executable by the MCU. Thus, one needs to extract the machine code from it in the Intel Hex format

   avr-objcopy -O ihex -R .eeprom blink.elf blink.ihex

Notes:

- You can see the detailed compilation process by enabling -v (verbose) option. For example,

  avr-gcc -v -mmcu=attiny13 -o blink.bin blink.c

- You can Generate all intermidiate files

  avr-gcc -mmcu=attiny13 -save-temps blink.c

- You should always enable optimization with the -Os parameter

  avr-gcc -v -Os -mmcu=attiny13 -save-temps blink.c

Building automation

The need for building automation

- Building can be tedious

The need for building automation

- Building can be tedious
  - Many GCC flags

The need for building automation

- Building can be tedious
  - Many GCC flags
  - Many independent commands

The need for building automation

- Building can be tedious
  - Many GCC flags
  - Many independent commands
  - Many build targets

The need for building automation

- Building can be tedious
  - Many GCC flags
  - Many independent commands
  - Many build targets
  - Many supported architectures

The need for building automation

- Building can be tedious
  - Many GCC flags
  - Many independent commands
  - Many build targets
  - Many supported architectures
  - Many source files

The need for building automation

- Building can be tedious
  - Many GCC flags
  - Many independent commands
  - Many build targets
  - Many supported architectures
  - Many source files
- Building manually can cause consistency issues waste development time

The need for building automation

- Building can be tedious
  - Many GCC flags
  - Many independent commands
  - Many build targets
  - Many supported architectures
  - Many source files
- Building manually can cause consistency issues waste development time

- Real world software is complex. For example, the Linux kernel contains:

The need for building automation

- Building can be tedious
  - Many GCC flags
  - Many independent commands
  - Many build targets
  - Many supported architectures
  - Many source files
- Building manually can cause consistency issues waste development time

- Real world software is complex. For example, the Linux kernel contains:
  - More than 23,000 .c files

The need for building automation

- Building can be tedious
  - Many GCC flags
  - Many independent commands
  - Many build targets
  - Many supported architectures
  - Many source files
- Building manually can cause consistency issues waste development time
- Real world software is complex. For example, the Linux kernel contains:
  - More than 23,000 .c files
  - More than 18,000 header file

The need for building automation

- Building can be tedious
  - Many GCC flags
  - Many independent commands
  - Many build targets
  - Many supported architectures
  - Many source files
- Building manually can cause consistency issues waste development time

- Real world software is complex. For example, the Linux kernel contains:
  - More than 23,000 .c files
  - More than 18,000 header file
  - More than 1,400 assembly files

The need for building automation

- Building can be tedious
  - Many GCC flags
  - Many independent commands
  - Many build targets
  - Many supported architectures
  - Many source files
- Building manually can cause consistency issues waste development time
- Real world software is complex. For example, the Linux kernel contains:
  - More than 23,000 .c files
  - More than 18,000 header file
  - More than 1,400 assembly files
  - How would you compiler this manually?

The need for building automation

- Building can be tedious
  - Many GCC flags
  - Many independent commands
  - Many build targets
  - Many supported architectures
  - Many source files
- Building manually can cause consistency issues waste development time
- Real world software is complex. For example, the Linux kernel contains:
  - More than 23,000 .c files
  - More than 18,000 header file
  - More than 1,400 assembly files
  - How would you compiler this manually?
- In most cases, one can use an Integrated development environment (IDE) to automate this process.

The need for building automation

- Building can be tedious
  - Many GCC flags
  - Many independent commands
  - Many build targets
  - Many supported architectures
  - Many source files
- Building manually can cause consistency issues waste development time

- Real world software is complex. For example, the Linux kernel contains:
  - More than 23,000 .c files
  - More than 18,000 header file
  - More than 1,400 assembly files
  - How would you compiler this manually?
- In most cases, one can use an Integrated development environment (IDE) to automate this process.

The need for building automation

- Building can be tedious
  - Many GCC flags
  - Many independent commands
  - Many build targets
  - Many supported architectures
  - Many source files
- Building manually can cause consistency issues waste development time

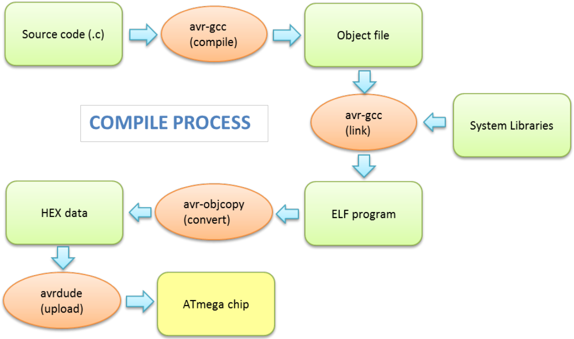
- Real world software is complex. For example, the Linux kernel contains:
  - More than 23,000 .c files
  - More than 18,000 header file
  - More than 1,400 assembly files
  - How would you compiler this manually?
- In most cases, one can use an Integrated development environment (IDE) to automate this process.

Why use an automatic build system?

Build Management Software (or Build Automation) provides a simple and consistent method for producing a target executable

Build Management Software

- Automated the process of
  - Preprocessing
  - Assembling
  - Compiling
  - Linking
  - Relocating
  - Upload the machine code to the microcontroller
- GNU Toolset performs all operations using make
- Real world make files are complex[1], but are often preferred to using IDE[2]



[1] https://www.gnu.org/software/make/manual/html_node/Complex-Makefile.html
[2] https://www.embeddedrelated.com/showthread/comp.arch.embedded/252000-1.php

Example make file

```
FILENAME    = blink
PORT        = /dev/cu.usbserial-00000000
DEVICE      = attiny13
PROGRAMMER  = arduino
BAUD        = 115200
COMPILE     = avr-gcc -Wall -Os -mmcu=$(DEVICE)


default: compile upload clean

compile:
	$(COMPILE) -c $(FILENAME).c -o $(FILENAME).o
	$(COMPILE) -o $(FILENAME).elf $(FILENAME).o
	avr-objcopy -j .text -j .data -O ihex $(FILENAME).elf $(FILENAME).hex
	avr-size --format=avr --mcu=$(DEVICE) $(FILENAME).elf

upload:
	avrdude -v -p $(DEVICE) -c $(PROGRAMMER) -P $(PORT) -b $(BAUD) -U flash:w:$(FILEN

clean:
	rm $(FILENAME).o
	rm $(FILENAME).elf
	rm $(FILENAME).hex
```

The end