

Embedded Systems

Embedded system toolchain

Lesson 11

Francesco Menichelli

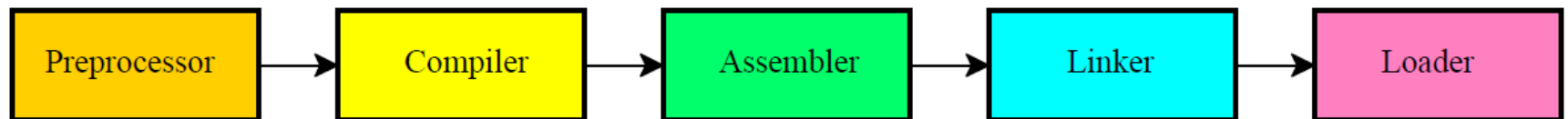
francesco.menichelli@uniroma1.it

Introduction

- A software toolchain is a set of programs invoked sequentially to turn source code into executable machine code.
- In an embedded system, the end result is machine code that is written into on-chip nonvolatile memory, such as Flash memory.
- The tool chain consists of many programs such as compilers, assemblers, linkers.
- Building software applications for embedded systems requires good knowledge of the development tools as well as the targeted hardware.

Overview

- Tool chains are vendor and hardware dependent, but a number of fundamental concepts are common to most development tool chains.
- All tool chains translate software from source code (e.g. a high-level language such as C) to machine code. An example tool chain is shown below:



Toolchain components

- Preprocessor
 - Performs several mechanical operations to prepare a source file for the *compiler*.
 - macro processing (`#define`)
 - selecting source text for compilation (`#ifdef`)
 - incorporating different shared files (`#include`)
- Compiler
 - The compiler is a tool for translating programs into a variety of forms, one form is *assembly language*.
 - More specifically, the compiler for an embedded system is usually a *cross compiler* that runs on one machine, called the *development platform* (e.g. a PC), and generates code for a different machine, called the *target machine*.

Toolchain components

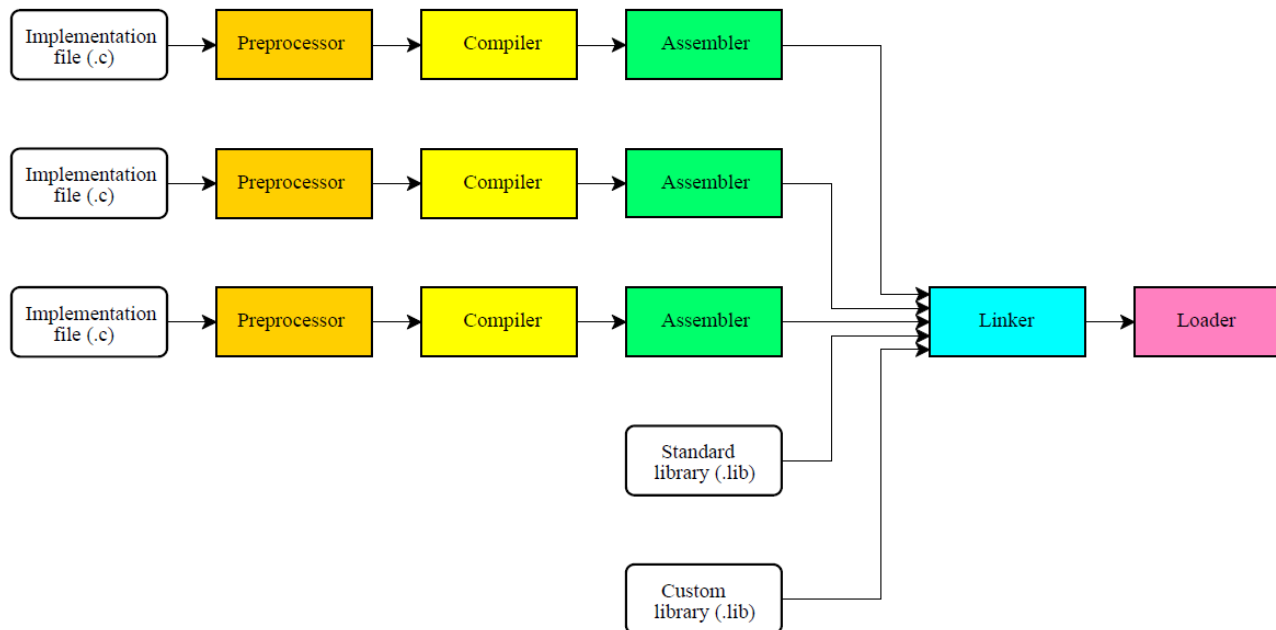
- Assembler
 - Convert the collection of assembly language modules produced by the compiler into machine language or object code.
 - An assembler normally produces relocatable code, which means that all memory addresses of both code and variables are as yet unknown.
- Linker
 - Combine all of the object code into a single file and resolve addresses (or identify address problems).
 - All addresses are now absolute.
 - The object code can come from user-written modules or library modules.
 - Library modules are collections of object code to provide standard C functionality (such as string handling, 32-bit operations on a 8-bit device, floating-point emulation, etc.) and specialized functionality

Toolchain components

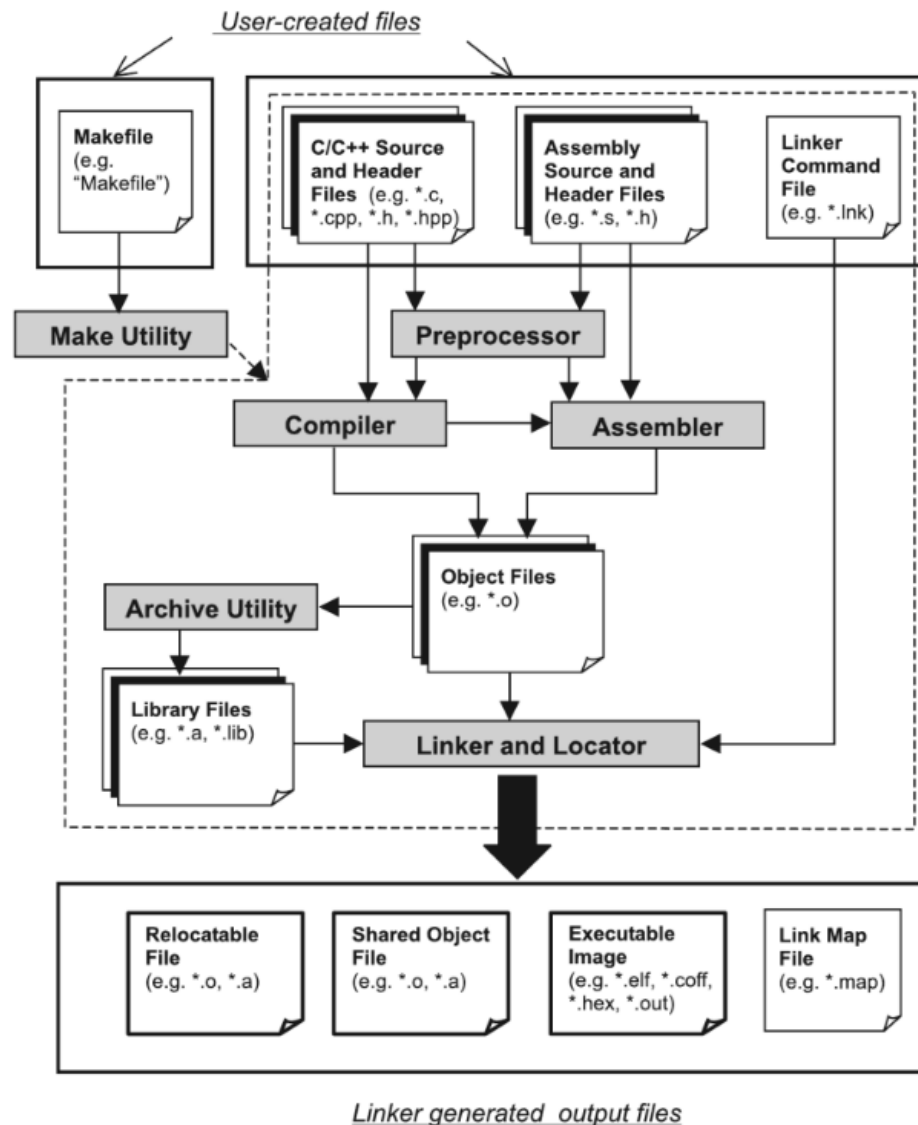
- Programmer
 - The loader has the job of translating the absolute object code created by the linker into a form that is suitable for downloading to a hardware programmer or debugger
 - The *Motorola S-record* and the *Intel HEX format* are two output formats which have been developed with the programming of non-volatile memory in mind.

Building a Software Project

- Embedded applications rely on fairly large pieces of software and therefore a modular approach to the software is generally taken.
- Typically, each piece of software is developed as a separate “implementation” C file (.c) with an associated “interface” header file (.h).
- The final program needs to bring together all the implementation files, header files, library files and other information in a process known as a “build”.



Building a Software Project



The GNU toolchain

- GNU make: Automation tool for compilation and build;
- GNU Compiler Collection (GCC): Suite of compilers for several programming languages;
- GNU Binutils: Suite of tools including linker, assembler and other tools;
- GNU Debugger (GDB): Code debugging tool;

GNU make

- A utility that automatically builds executable programs and libraries from source code by reading files called makefiles which specify
- Make can be used to manage any project where some files must be updated automatically from others whenever the others change.
- A makefile consists of rules. Each rule begins with a textual dependency line which defines a target followed by a colon (:) and optionally an enumeration of components (files or other targets) on which the target depends. The dependency line is arranged so that the target (left hand of the colon) depends on components (right hand of the colon). It is common to refer to components as prerequisites of the target.

GNU make

```
CFLAGS ?= -g
```

```
all: helloworld
```

```
helloworld: helloworld.o
```

```
    # Commands start with TAB not spaces
```

```
    $(CC) $(LDFLAGS) -o $@ $^
```

```
helloworld.o: helloworld.c
```

```
    $(CC) $(CFLAGS) -c -o $@ $<
```

```
clean: FRC
```

```
    rm -f helloworld helloworld.o
```

```
    # This pseudo target causes all targets that depend on FRC
```

```
    # to be remade even in case a file with the name of the target  
exists.
```

```
    # This works with any make implementation under the assumption  
that
```

```
    # there is no file FRC in the current directory.
```

```
FRC:
```

GNU Compiler Collection

- Users invoke a language-specific driver program (gcc for C, g++ for C++, etc.), which interprets command arguments, calls the actual compiler, runs the assembler on the output, and then optionally runs the linker to produce a complete executable binary.
- `gcc -S main.c` (compile, produce main.s)
- `gcc -c -o main.o main.c` (compile and assemble, produce main.o)
- `gcc -o program main.c` (compile, assemble, link, produce “program” executable)

Gcc compile

- gcc -S main.c

arm-none-eabi-gcc -marm -mcpu=arm7tdmi -S main.c

main.c

```
void f1(void);
void f2(void);

volatile int g1;
int main(void)
{
    volatile int l1,l2;

    l1=1;
    l2=2;
    g1=3;
    f1();
    f2();
}

void f1(void) {
    volatile int a;
    a++;
}
```

main:

```
stmfd    sp!, {fp, lr}
add      fp, sp, #4
sub      sp, sp, #8
mov      r3, #1
str      r3, [fp, #-8]
mov      r3, #2
str      r3, [fp, #-12]
ldr      r3, .L2
mov      r2, #3
str      r2, [r3]
bl       f1
bl       f2
mov      r0, r3
sub      sp, fp, #4
ldmfd    sp!, {fp, lr}
bx       lr
```

.L2:

```
.word    g1
```

main.s

Gcc compile and assemble

- `gcc -c -o main.o main.c`

`arm-none-eabi-gcc -marm -mcpu=arm7tdmi -c main.c`

`arm-none-eabi-objdump -d main.o > main.o.dis`

main.c

```
void f1(void);
void f2(void);

volatile int g1;
int main(void)
{
    volatile int l1,l2;

    l1=1;
    l2=2;
    g1=3;
    f1();
    f2();
}

void f1(void) {
    volatile int a;
    a++;
}
```

main.o.dis

```
00000000 <main>:
0:      e92d4800  push    {fp, lr}
4:      e28db004  add     fp, sp, #4
8:      e24dd008  sub     sp, sp, #8
c:      e3a03001  mov     r3, #1
10:     e50b3008  str     r3, [fp, #-8]
14:     e3a03002  mov     r3, #2
18:     e50b300c  str     r3, [fp, #-12]
1c:     e59f301c  ldr     r3, [pc, #28] ; 40 <main+0x40>
20:     e3a02003  mov     r2, #3
24:     e5832000  str     r2, [r3]
28:     ebfffffe  bl      44 <f1>
2c:     ebfffffe  bl      0 <f2>
30:     e1a00003  mov     r0, r3
34:     e24bd004  sub     sp, fp, #4
38:     e8bd4800  pop     {fp, lr}
3c:     e12fff1e  bx      lr
40:     00000000  .word   0x00000000
```

Gcc compile, assemble and link

- gcc -o program main.c

arm-none-eabi-gcc -marm -mcpu=arm7tdmi -o program main.c f2.c

arm-none-eabi-objdump -d program > program.dis

main.c

```
void f1(void);
void f2(void);

volatile int g1;
int main(void)
{
    volatile int l1,l2;

    l1=1;
    l2=2;
    g1=3;
    f1();
    f2();
}

void f1(void) {
    volatile int a;
    a++;
}
```

000081cc <main>:

```
81cc: e92d4800
81d0: e28db004
81d4: e24dd008
81d8: e3a03001
81dc: e50b3008
81e0: e3a03002
81e4: e50b300c
81e8: e59f301c
81ec: e3a02003
81f0: e5832000
81f4: eb000005
81f8: eb00000d
81fc: e1a00003
8200: e24bd004
8204: e8bd4800
8208: e12fff1e
820c: 00010b20
```

program.dis

```
push    {fp, lr}
add     fp, sp, #4
sub     sp, sp, #8
mov     r3, #1
str     r3, [fp, #-8]
mov     r3, #2
str     r3, [fp, #-12]
ldr     r3, [pc, #28];
mov     r2, #3
str     r2, [r3]
bl      8210 <f1>
bl      8234 <f2>
mov     r0, r3
sub     sp, fp, #4
pop     {fp, lr}
bx      lr
.word   0x00010b20
```

GNU Binutils

- The GNU Binary Utilities, or binutils, are a set of programming tools for creating and managing binary programs, object files, libraries, profile data, and assembly source code
- Code production
 - as assembler
 - ld linker
 - ar create, modify, and extract from archives
- Code analysis
 - nm list symbols in object files
 - objcopy copy object files, possibly making changes
 - objdump dump information about object files

GNU C Library

- The GNU C Library, commonly known as glibc, is the GNU Project's implementation of the C standard library. Despite its name, it now also directly supports C++
- The C standard library provides macros, type definitions, and functions for tasks like string handling, mathematical computations, input/output processing, memory allocation and several other operating system services.
- glibc is a library written for general purpose systems (large memory available)
- Several alternative C standard libraries have been created which emphasize a smaller memory footprint.
 - Bionic (used in Android)
 - uClibc
 - Newlib
 - EGLIBC

ABI (application binary interface)

- When compiling programs that run on their own on a processor (“bare metal” system), it is required a compiler that produces instructions for that particular processor.
- When compiling programs that interact with other programs, then all the programs involved need to be able to communicate.
 - Making system calls
 - Making library calls
- This case requires that the caller and the callee agree on the binary representation of data
 - e.g. endianness
 - how to pass data
 - which registers to use for function arguments
 - how function calls affect the stack
- The specification of how this works is called an **ABI** (application binary interface).

ABI (application binary interface)

- Adhering to ABIs (which may or may not be officially standardized) is usually the job of the compiler, OS or library writer, but application programmers may have to deal with ABIs directly when writing programs in a mix of programming languages, using foreign function call interfaces between them.

ABI (application binary interface)

- ABIs cover details such as:
 - the sizes, layout, and alignment of data types
 - the calling convention, which controls how functions' arguments are passed and return values retrieved;
 - for example, whether all parameters are passed on the stack or some are passed in registers, which registers are used for which function parameters, and whether the first function parameter passed on the stack is pushed first or last onto the stack
- how an application should make system calls to the operating system and
- and in the case of a complete operating system ABI, the binary format of object files, program libraries and so on.

EABI (Embedded ABI)

- An embedded-application binary interface (EABI) specifies standard conventions for file formats, data types, register usage, stack frame organization, and function parameter passing of an embedded software program.
- Compilers that support the EABI create object code that is compatible with code generated by other such compilers, allowing developers to link libraries generated with one compiler with object code generated with another compiler.
- Developers writing their own assembly language code may also use the EABI to interface with assembly generated by a compliant compiler.
- The main differences between an EABI and an ABI for general-purpose operating systems are that
 - privileged instructions are allowed in application code
 - dynamic linking is not required (sometimes it is completely disallowed)
 - a more compact stack frame organization is used to save memory

The gcc naming convention

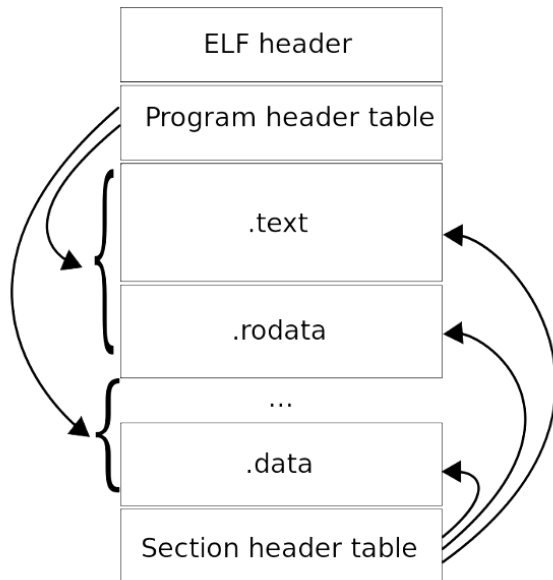
- The general naming convention for GCC cross-compilers is
 - CPU-VENDOR-SYSTEM-gcc
- Cross-compilers for ARM are usually called
 - arm-VENDOR-OS-LIBC-ABI-gcc with some of the parts omitted.
- For example
 - arm-acme-linux-gnu-gnueabi-gcc would be GCC for ARM (v7 by default), for ACME's distribution of Linux, set to link against Glibc, using the gnueabi ABI.
 - arm-linux-gnu-gcc indicates a compiler set to link against Glibc on Linux, with an unspecified ABI.
 - arm-linux-gcc doesn't specify which libc is targeted.
 - arm-none-linux-gnueabi (CodeSourcery ARM compiler for linux)
 - arm-none-eabi (CodeSourcery ARM compiler for bare-metal systems)

ELF Binary output format

- Executable and Linkable Format (ELF, formerly called Extensible Linking Format) is a common standard file format for executables, object code, shared libraries
- Each ELF file is made up of one ELF header, followed by file data. The file data can include:
 - Program header table, describing zero or more segments
 - Section header table, describing zero or more sections
 - Data referred to by entries in the program header table or section header table

ELF format

- An *ELF header* resides at the beginning and holds a "road map" describing the file's organization.
 - *Sections* hold the bulk of object file information for the linking view: instructions, data, symbol table, relocation information, and so on.
 - A *program header table*, if present, tells the system how to create a process image. Files used to build a process image (execute a program) must have a program header table
 - A *section header table* contains information describing the file's sections. Every section has an entry in the table; each entry gives information such as the section name, the section size, and so on.



An ELF file has two views: The program header shows the segments used at run-time, whereas the section header lists the set of sections of the binary.

ELF sections

- Name: name of the section
- Size: size in bytes
- VMA: section virtual address
- LMA: section physical address
- File offset: position of the section inside elf file
- Align: byte alignment
- Flags: attributes

Sections:

Idx	Name	Size	VMA	LMA	File off	Algn
0	.text	000000f8	00000000	00000000	00008000	2**2
	CONTENTS, ALLOC, LOAD, READONLY, CODE					
1	.rodata	00000004	000000f8	000000f8	000080f8	2**2
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
2	.data	00000018	a0000000	000000fc	00010000	2**2
	CONTENTS, ALLOC, LOAD, DATA					
3	.bss	00000004	a0000018	00000114	00010018	2**2
	ALLOC					
4	.comment	0000005b	00000000	00000000	00010018	2**0
	CONTENTS, READONLY					
5	.ARM.attributes	0000002e	00000000	00000000	00010073	2**0
	CONTENTS, READONLY					

ELF section address







- Every loadable or allocatable output section has two addresses.
 - VMA, or virtual memory address. The address the section will have when the output file is run.
 - LMA, or load memory address. This is the address at which the section will be loaded.
- In most cases the two addresses will be the same. An example of when they might be different is when a data section is loaded into ROM, and then copied into RAM when the program starts up
- In this case the ROM address would be the LMA, and the RAM address would be the VMA

ELF Section Attributes

- Alloc (ALLOC)
 - is loaded into memory at runtime. This is true for code and data sections, and false for metadata sections.
- Exec (CODE)
 - has permission to be run as executable code.
- Write (READONLY)
 - is writable at runtime
- Progbits (LOAD)
 - is stored in the disk image, as opposed to allocated and initialized at load.
- Align=n
 - requires a memory alignment of n bytes. The value n must always be a power of 2.

Other executable formats

- Raw (binary)
 - Reproduces the exact image of the physical memory, including empty parts
- Intel Hex
 - conveys binary information in ASCII text form.
 - consists of lines of ASCII text that are separated by line feed or carriage return characters or both.
 - Each text line contains hexadecimal characters that encode multiple binary numbers.
 - The binary numbers may represent data, memory addresses, or other values, depending on their position in the line and the type and length of the line.
 - Each text line is called a record.

 Start code  Byte count  Address  Record type  Data  Checksum

```
:10010000214601360121470136007EFE09D2190140
:100110002146017E17C20001FF5F16002148011928
:10012000194E79234623965778239EDA3F01B2CAA7
:100130003F0156702B5E712B722B732146013421C7
:00000001FF
```

Example shows a file that has four data records followed by an end-of-file record

Other executable formats

- Motorola Srec
 - conveys binary information in ASCII text form.
 - consists of lines of ASCII text that are separated by line feed or carriage return characters or both
 - The lines are called srecords

 Record type  Byte count  Address  Data  Checksum

```
S00F000068656C6C6F202020202000003C
S11F00007C0802A6900100049421FFF07C6C1B787C8C23783C6000003863000026
S11F001C4BFFFFE5398000007D83637880010014382100107C0803A64E800020E9
S111003848656C6C6F20776F726C642E0A0042
S5030003F9
S9030000FC
```