
The Development System

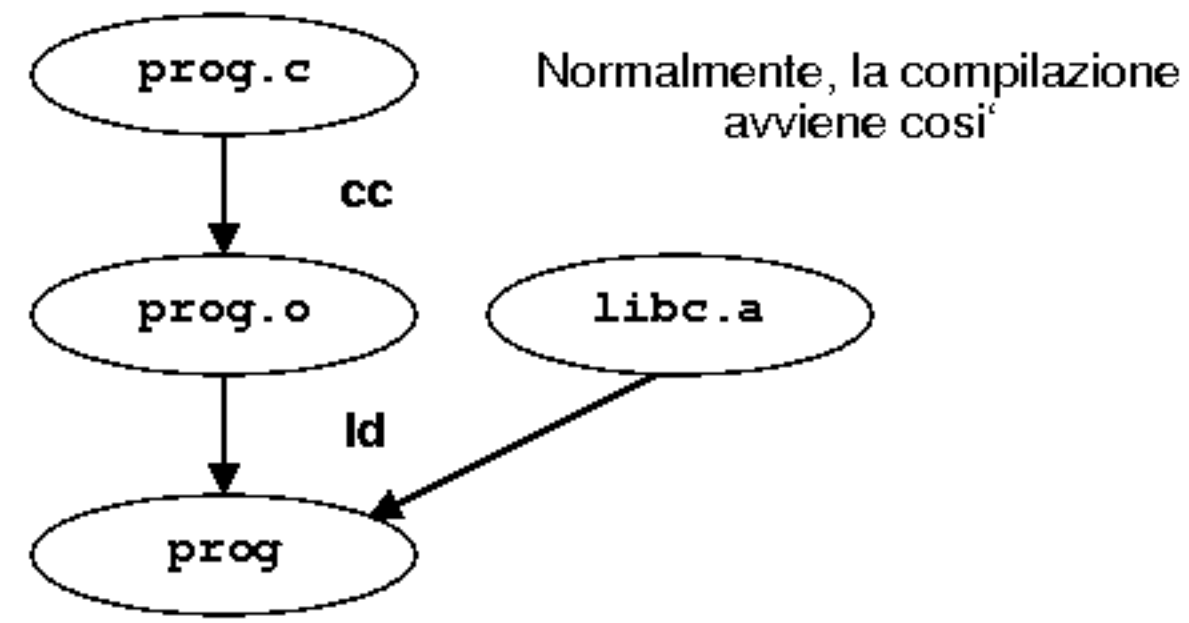
The Compiler and Linker

To compile a C source you perform 2 (or rather 4) steps

- **cc (gcc) converts a source file into an object file**
 - ♦ **cpp is the preprocessor**
 - ♦ **cc1 is the compiler proper**
 - ♦ **as is the assembler**
- **ld performs the final linking step**

For simple programs, you can forget about those steps

- `gcc prg.c -o prg`



Static and Dynamic Libraries

Libraries can be static (.a) or dynamic (.so)

- **the .a suffix means "archive"**
 - ♦ **such files are generated by "ar"**
 - ♦ **it's a very simple format**
 - ♦ **the archives are not bound to host executable files**
 - ♦ **"tar" is similar: it is the "tape archive"**
- **the .so suffix means "shared object"**
 - ♦ **the files are generated by "ld" (or the gcc frontend)**
 - ♦ **the format is ELF, they are not generic archives**
- **Static libraries are easier to use**
- **Static libraries make debugging easier**
- **Dynamic libraries require less memory**
- **Using dynamic libraries in custom projects is not trivial**

Executable Formats

A "binary format" is a file format for programs

- **An executable is generally made up of three parts**
 - ♦ **.text: code proper**
 - ♦ **.data: initialized data**
 - ♦ **.bss: zeroed data**
- **Only the first two are saved to disk**
- **The "size" command shows the size of the program**

ELF (executable and loadable format) offers:

- **Arbitrary section names**
- **Any number of sections**
- **Definition of object files, executables, libraries**

Compiler and linker can build ELF files built as you like

- **Especially, "ld" can be configured with an "ldscript"**

uClinux has its own binary format ("flat")

Make

The "make" program is used to "make" files

- It is configured by "makefile" or "Makefile"
- The makefile is written in a descriptive language
- Execution of each rule is based on file date and time
- It uses implied rules on file names

GNU make has some procedural features as well

- Conditional execution
 - ♦ ifdef
 - ♦ ifeq
- Immediate assignment (":=")
- Incremental assignment ("+=")
- Conditional assignment ("?=")

The Command Line of Make

The command "make sth" actually makes sth.

- **"make -n sth" ("not") does not make, but it shows what it would make**
- **"make -k sth" ("keep going") proceeds in case of error**
- **"make -j<n> sth" ("jobs") runs in parallel (for SMP)**
- **"make sth VAR=val" makes sth assigning "val" to "VAR"**

There are some conventional targets

- **"make all" compiles**
- **"make install" installs**
- **"make clean" cleans the source tree (removing compiled code)**
- **"make distclean" makes things as clean as distributed**

To know whether or not to make sth, make uses the date/time of all prerequisite files

Make Variables

Make variables come from different sources

- the command line
- the makefile
- environment variables

Make uses a number of predefined variables

- CC, CFLAGS (compiler and arguments)
- LD, LDFLAGS (linker and arguments)
- @, *, <, ^ (target, stem, dep.)
- MAKE (make and its own arguments)

Variables are only expanded when used

- Unless you use ":", but please don't overdo

This is a simple makefile

```
CFLAGS = -O2 -g -Wall  
all: prog1 prog2
```

Make Rules

A makefile is a set of dependency rules

```
target: dep [dep ...]  
    cmd args  
    cmd2 args2
```

Some targets can be defined as phony targets

```
.PHONY: all clean distclean
```

- Rule concatenation is automatic
- Commands must return 0 if they succeed
- When an error occurs, the graph is broken
- With good makefiles, "make -j" works pretty well

Implicit Rules

Rules for common operations are predefined

You can define new implicit rules

```
%.old: %.c
        cp $*.c $*.old
```

```
SRC = $(wildcard *.c)
```

```
old: $(SRC:.c=.old)
```

"make -d" prints a lot of debugging information

Make and Integrated Environments

From within emacs, you can control make

- "M-x compile" runs an external command ("make -k" by default)
- "C-x `" (next-error) parses the command's output
- Grep can be called similarly ("M-x grep")

The same mechanism is used by all other IDEs

- They call an external process and capture its stdout/stderr
- They run a regexp on captured strings, to find errors
- They know of success/failure from the process' return value
- Every program should report errors in the same way, for interoperability
 - ♦ A good simple way is: <prgname>: <inputfile>: <line>: <error>

Objdump

To analyze object and executable files, there is objdump

- ♦ **to disassemble: `objdump -dr <file>`**
- ♦ **to see ELF headers: `objdump -h <file>`**
- ♦ **to see an ELF section: `objdump --full-contents`**
- ♦ **to see source and assembly: `objdump -S`**

Objdump allows looking in binary files too

- ♦ **`objdump -b binary -m arm -D <file>`**
- ♦ **`objdump -b binary -m i386 --adjust-vma=<addr> -D <file>`**
- **We can thus look at boot loader code**
- **We can look at a peripheral's firmware image**

Inline assembly

gcc allows inline assembly in C sources or headers

- The code must interact with the optimizer
- The syntax is not trivial at all

```
asm("code" : r-output : r-input : r-clobber);
```

The "code" string can't use explicit register names

- You can use positional names ("%0") or symbolic names ("%[timeout]")

The register lists (out and in) declare their C expressions

The list of clobbered registers can also include

- memory: it means memory external to the CPU has been modified
- cc: it means "condition code" flags are modified by the asm code

Examples:

```
#define set_cr(x) /* arm */                \
    __asm__ __volatile__(                  \
        "mcr    p15, 0, %0, c1, c0, 0    @ set CR" \
        : : "r" (x) : "cc")
```

```
#define mb() __asm__ __volatile__ ("" : : : "memory")
```

```
#define rdtsc(low,high) /* x86 */      asm("rdtsc" : "=a" (low), "=d" (high))
```

All documentation is part of the gcc manual