The Development System

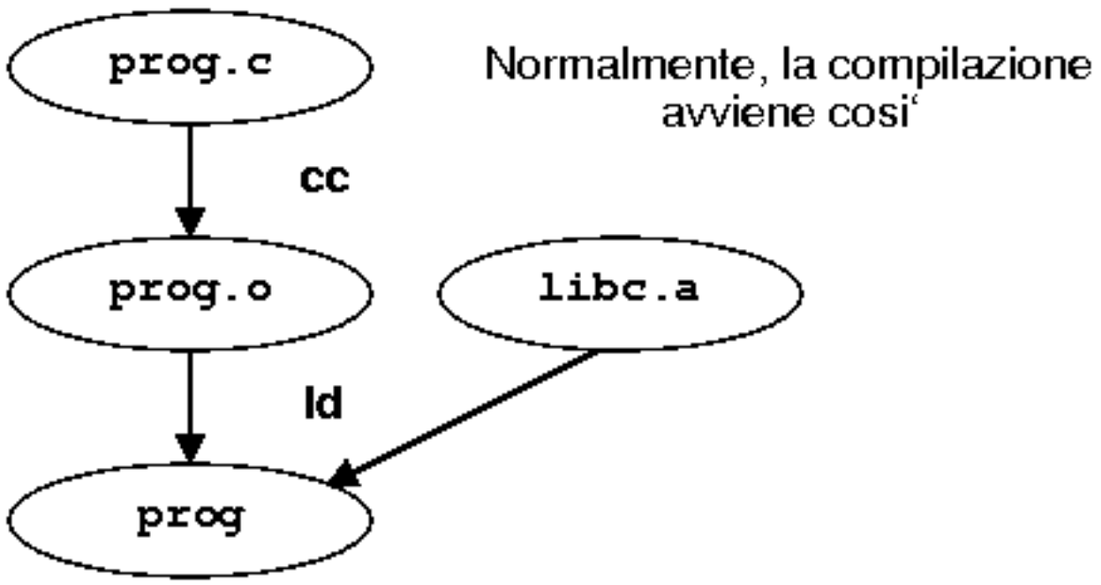
The Compiler and Linker

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 con 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 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

Cross-compilation

Cross Compilation: GNU/Linux Conventions

A cross-compiler is a compiler that creates executable code for another processor

Building by yourself cross-code requires three parameters

- **--build** (where is the code being built -- usually autodetected)
- **--host** (where will the code be hosted)
- **--target** (where will the code run)

Cross-compilation tools usually have a prefix in the filename

- **arm-linux-gcc arm-buildroot-linux-gnueabi-gcc**
- **m68k-linux-gcc lm32-elf-gcc**

Building the kernel:

```
make CROSS_COMPILE=m68k-linux- ARCH=m68knommu
```

Creating a Cross Compiler

To cross-compile, you need three packages

- binutils (assembler, linker)
- gcc (the compiler proper)
- glibc or another libc implementation
  - As a minimal fallback, you can use "newlib"

The steps to build the compiler are:

- compiling binutils
- compiling the bootstrap gcc
- compiling libc
- compiling the final gcc (with C++, Java etc)

Most embedded distributions nowadays build the compiler first

Still, you can build your own, especially for uC targets

Pre-Built and Crosstool-ng

A few developers or companies offer pre-built toolchains

- Unfortunately, the list is very volatile
- Refer to elinux for a "current" list
  - https://elinux.org/Toolchains

Another option is crosstool-ng

- Much more difficult than the original "crosstool"
- Kconfig based
- Reported to be reliable

For bare-metal (or kernel, or bootloader), distributions help

- A suitable arm-none-eabi-gcc is usually packaged
- Check your distribution for details

Please note the naming: "arm-none-eabi"

- The first word is the CPU family
- The second word is the host operating system
- The later (optional) words are variants
- We'll talk about ARM and EABI later on

Programs Included in Binutils

Tool-chain proper commands

- as
- ld

Maintainance commands

- strip
- objcopy

Information retrivial commands

- objdump, nm
- size, strings

Most of the information is abstracted to libbfd

- The library offers An API to read/write an object file
- You can create a multi-platform objcopy
- Some distributions offer "binutils-multiarch"
  - It lacks AVR or LM32 support, but most CPU families are there

Cross Compiling Applications

Simple packages:

```
make CC=my-cross-gcc
```

Autotools packages:

```
CC=my-cross-gcc ./configure --prefix=/usr --host=arm-linux \
            && make && make install DESTDIR=/target
```

Special case: gdbserver (tested w/ version 9.1):

```
CC=my-cross-gcc /path/to/gdb-9.1/gdb/gdbserver/configure \
            --host=arm-linux --prefix=/usr \
             && make && make install DESTDIR=/target
```

Special case: gdb (tested w/version 9.1):

```
/path/to/gdb-9.1/configure --target=arm-linux \
            --prefix=/usr/local/cross-tools \
            && make && make install
```

Cross Compiling Bare-Metal systems

Most bare-metal systems follow the Linux tradition:

- The "CROSS_COMPILE" variable sets the prefix
- The "ARCH" variable sets the top-level subtree

Also, most bare-metal systems are based on Kconfig

- You can "make defconfig" (depends on $ARCH)
- Mode defaults live in the configs/ subdir
- You can interactively change the configuration
  - "make config", "make oldconfig", "make menuconfig", ...
- The current configuration lives in .config (the output of configuring)
  - This is an input file for make
  - A corresponding header is generated for C language

One problem with Kconfig is reproducing a build

- If it can fit, it makes sense to save .config in the binary
- This in addiion to the build commit, a mandatory item

Please consider setting up reproducible builds for your own OS