

djyos si 版本 makefile 说明

本文件适用 djyosV0.4.1 及以后版本。

1. djyos 的 makefile 的特点

- 1、直观易读，目标依赖关系很直观，深度最多两级，没有多重依赖转来转去让人摸不着头脑的地方。多用可读单词，非不得已不用“`^$@?+*<%`”等天书一般的符号。
- 2、自动化程度高，任何目录下增加删除改名文件或子目录，只需修改该目录下的 makefile 文件即可，不但总控 makefile 在内的所有其他 makefile 都不用修改，就连连接脚本.ld 文件也自动生成。
- 3、应用程序增减源程序文件和目录时，当前目录下的 makefile 需修改的部分也很少。可能要改的只有 6 行，实际上可能只有 1 行需要改。

2. 使用 makefile

使用 makefile 之前，你必须先建立 gcc 的编译环境，可以按照与本文同时发布的《建立 windows 下 djyos for arm 的编译和调试环境.doc》，建立 windows 下的 gcc 编译环境。

2.1. makefile 的功能

make 就是制作的意思，而 makefile 文件就是定义了一系列制作目标的规则。make 的目的，就是在 makefile 中描写的规则的指引下，生成最终所需要的目标，什么是最终目标呢？最终目标是执行 make 命令时从命令行输入的：

```
make clean
```

clean 就成了最终目标。怎样才算生成了最终目标呢？我们看看 makefile 中生成 clean 目标的规则：

```
clean:$(subdir)
```

```
    rm -f *.ld ld_obj_list
```

可以看到，clean 目标依赖\$(subdir)目标，下面还有一条 rm 命令，也就是说，如果\$(subdir)目标成功生成，并且 rm 命令成功执行，clean 目标就算完成了。

又比如命令行输入

```
make debug
```

最终目标 target 的值将等于 debug，我们看 makefile 中对 target 目标的描述：

```
$(target): rm_obj_list $(subdir)
```

```
$(sub_make) -f make_ld $(MAKECMDGOALS)
```

target 目标依赖 rm_obj_list 和\$(subdir)两个目标（他们的顺序不能颠倒），这两个目标成功生成，并且：

```
$(sub_make) -f make_ld $(MAKECMDGOALS)
```

命令成功执行的话，就算 debug 目标成功生成了。

2.2. 编译生成的文件说明

*.elf: elf 格式的可执行文件，包含符号表，还可包含更多的调试信息。

*.bin: 可直接烧录的二进制可执行文件。

*.ld: 连接脚本文件, 用于控制连接过程, 描述目标代码存储与运行地址。

ld_obj_list: 用于生成*.ld 文件的临时文件。这个文件在编译结束后即删除, 如果用户在修改 makefile 过程中需要查看这个文件来定位错误, 可在 make_ld 文件中注释掉删除本文件的语句。

2.3. 编译命令和目标说明

在命令行输入下列命令可以编译不同的目标文件:

1、 make boot_rom 命令

编译产生 boot_rom.bin 文件, 这个文件只包含启动代码, 然后就进入死循环。当需要用硬件仿真器调试操作系统时, 就应该先编译 boot_rom.bin 并烧录到 0 地址的 norflash 中。

2、 make debug 命令

编译产生产生用于调试的 debug.elf 文件, 编译器的优化级别是 0, 且含 gdwarf-2 格式的调试信息, 由调试软件比如 realview2.2 或 gdb 直接加载到内存中调试。

3、 make run_inram 命令

编译产生可直接烧录到 0 地址的 norflash 中运行的二进制可执行文件 run_inram.bin, 由于经过编译器优化, 虽然同时产生 elf 格式的可执行文件, 但调试不方便。本命令产生的可执行文件执行时会自动把代码 copy 到 ram 中, 自动执行内存清零操作, 然后跳转到 ram 中执行, si 版本的 djyos 没有独立的 bootloader, 可近似理解为 si 版本内含 bootloader 的部分功能。

4、 make run_inflash 命令

本命令产生 run_inflash.bin 文件, 与 run_inram 命令相似, 唯一不同的是, 本目录生成的可执行文件不把代码 copy 到 ram 中, 而是直接在 flash 中运行, 适合于在 ram 很少的单片机中使用。

5、 make clean 命令

删除除.bin 和.elf 外的所有由编译产生的文件。

2.4. 何时应该使用 make clean

make 有一个特点, 就是如果被依赖的文件比目标文件更新的话, 就会重新编译目标文件, 否则不会重编译。比如两次编译间, a.c 文件没有被编辑过, 那么被依赖的文件 a.c 就不比目标文件 a.o 新, 因此第二次编译时 a.o 就不会被重新编译, 但有时候, 我们不希望这种“查新”机制起作用, 就要用 make clean 命令把目标文件删掉, 强制重新编译。

1、 修改头文件

原理上讲, 如果依赖关系做得好, 头文件也成为.o 文件的依赖文件的话, 头文件修改后, 编译器能正确重编译依赖该头文件的目标文件, 但事与愿违, 有些时候头文件与 c 源文件的依赖关系很复杂, 并且在编程过程中会有调整, 要完全描述这种依赖关系变得很困难, 且 makefile 会被弄得很复杂难读。因此, djyos 的 makefile 不描述头文件和 C 文件的依赖关系, 故在修改头文件后, 需要执行 make clean 命令删掉所有.o 文件, 强迫重新生成所有.o 文件。

2、 目标改变

比如原来执行过:

```
make debug 编译调试版本
```

现在要执行

make run_inram 编译可直接执行版本, 由于在执行 debug 时, 已经产生了.o 文件, 而.c 文件又没有修改, 故不会重新编译, 而是直接把编译 debug 时产生的.o 文件重新连接。而 run_inram 和 debug 的命令参数并不一样, 该.o 文件是按 debug 的命令参数编译的, 不适合 run_inram 使用, 必须重新按 run_inram 的参数产生所有.o 文件, 故需要在执行 make run_inram 之前用 make clean 删掉所有.o 文件。

3、源代码版本发布或归档

发布源代码时，实际上只希望发布源代码文件和.bin文件，就要用 `make clean` 删掉其他编译产生的文件。

-----如果你只是想知道如何编译 djyos，到此为止-----

3. 修改 makefile

当用户移植 djyos，或者添加应用程序或者 driver 时，就需要修改或建立新的子 makefile。

3.1. 修改总控 makefile

总控 makefile 在工程目录下，比如工程目录是 myprj，则在 myprj 目录下的那个 makefile 文件就是总控 makefile 文件了。

djyos 被移植到新的 cpu，或者改变指令集（由 ARM 改为 thumb），或者改变存储器大小端时，或者改变编译器名称时，就需要修改总控 makefile 文件。总控 makefile 中这些内容都用变量表示，语义非常简单明了，这里就不一一列出具体修改方法了。

总控 makefile 总共只有几十行，绝大部分在移植时是无需修改的，即使上述几个方面同时需要调整，需要修改的也不超过 10 行。一般情况下只需要修改：

1、CPU 发生变化

修改 `CPU = arm7tdmi` 这个变量

2、切换指令集

当移植到 flash 稀缺的 ARM 体系时，需要修改 `CFLAGS` 变量，加入 `-mthumb-interwork` 参数。

3、改变存储器配置

需修改 *.mld 文件中存储器描述的这几行：

```
MEMORY\n\
{\n\
\trom (rx) : org = 0,          len = 2M\n\
\ttram (wx) : org = 0x0c000000, len = 8M\n\
}\n\
```

还可能要修改设定中断向量地址的行：

```
\t_ISR_STARTADDRESS = 0x0c7fff00;    \n\
```

按用户板件的实际存储器配置修改即可。一共有 4 个 mld 文件，每个都要修改。

3.2. 添加和修改子 makefile

工程中添加子目录或者源程序文件时，就需要添加子 makefile，当改名、删除子目录或者源程序文件是，就需要修改子 makefile，在添加和修改过程中，makefile 中唯一需要修改的是以下几行：

```
subdir =
boot_rom =
rom_init =
preload =
sysload =
critical =
```

这几行涉及的变量含义参见第 5.1 节。这些变量的值，是由空格分隔的一个个字符串，注意无需双引号，行末也没有分号或其他符号，比如在 driver 目录下有两个包含源程序文件的子目录：uart、和 flash_chip，则 driver 目录下的 makefile 中 subdir 的值就这样描述：

```
subdir = uart flash_chip
```

要往 driver 目录下添加 tty 目录，则直接在后面添加 tty 就可以了，如下

```
subdir = uart flash_chip tty
```

然后 tty 目录下也要添加 makefile，只要从其他目录 copy 一个过来，再修改前述几个变量的值就可以了。

添加和删除以及改名源文件的方法相当简单，就是修改相应目录下的 makefile 中的 *boot_rom*、*rom_init*、*preload*、*sysload*、*critical* 中的一个或几个变量。

-----如果你不打算分析 djyos 的 makefile 实现方法，就此止步吧-----

4. 总控 makefile

总控 makefile 在工程目录下，比如工程目录是 myprj，则在 myprj 目录下那个 makefile 文件就是总控 makefile 文件了。实际上总控 makefile 文件有两个，分别是 makefile 和 make_ld 文件，为什么要分成两个呢？是为了实现自动化生成连接脚本文件，待介绍 make_ld 文件的时候再详细说明。下面开始介绍 makefile 的各部分代码，斜体部分是从 makefile 文件中 copy 的代码。

4.1. 变量

makefile 中有许多变量，这里简单说明一下。许多嵌入式系统开发者没有接触过 makefile，很容易用 C 语言的变量去理解 makefile 的变量。其实这两者的差别非常之大，C 语言的变量是一个地址索引，变量有类型，通过类型告诉操作系统如何理解该地址保存的数据。而 makefile 的变量就是字符串集，一个变量就是一个由空格隔开的一个个单词，连+*/这些符号也算单词的一部分，一句话，由空格分开的连续的字符构成一个单词。makefile 有许多方法处理变量，实际上处理的是字符串。makefile 中，这些字符串不需要用双引号。

1、topdir

```
export topdir = $(shell pwd)
```

工程根目录路径名字符串。export topdir = \$(shell pwd)的含义是，利用操作系统的 shell 命令取当前工作目录，由于总控 makefile 在工程根目录下，所以取得的是工程根目录的路径名，赋值给 topdir。export 的含义是使 topdir 在子 makefile 中可见，注意，在子 makefile 中，topdir 的值仍然是工程根目录串。

2、MCU

顾名思义，设置所使用的 cpu 核，比如在 44b0 版本就有：

```
MCU = arm7tdmi
```

3、sub_make

嵌套执行的 make 命令，变量值是：

```
export sub_make = @make --no-print-directory
```

@的含义是不在屏幕上输出本行命令，--no-print-directory 是不输出进入和离开某子目录的信息，这两者的使用使得屏幕输出比较简洁，当调试 makefile 时，可打开这两类输出，帮助定位错误。

4、MAKECMDGOALS

命令行引入的目标名称变量，比如执行 make boot_rom，该变量的值就是 boot_rom。

5、其他变量

target: 编译的终极目标名。

op: 编译器优化级别。

dbg: 产生调试信息，产生 debug 文件时需要，一般选 gdwarf-2。

CC: 编译器名，这里是 arm-elf-gcc，也可指定其他 gcc 编译器，比如 arm-eabi-gcc。

AS: 汇编器名，gcc 中汇编器和 c 编译器都使用 arm-elf-gcc。

objcopy: 从 elf 格式可执行文件提取二进制代码生成 .bin 文件的程序。

size: 显示 elf 文件中各部分尺寸

incdir: 设定编译器查找头文件的路径, 本变量要输出给 makefile 使用。

CFLAGS, ASFLAGS, LDFLAGS: 编译和连接时的命令行选项。

4.2. 生成目标

1、rm_obj_list 中间目标

删除编译子目录中产生的 ld_obj_list 文件, 这是一个临时文件, 如果上一次的编译过程成功完成, 本文件将被删除, 否则可能未删除。在执行新的编译之前, 必须重新生成 ld_obj_list, 故需要将老的删掉。

2、\$(subdir) 中间目标

顾名思义, 这个目标是进入子目录, 执行子目录下的编译任务, 在子目录的 makefile 中, 又以同样的命令执行子目录的编译任务, 就这样一级级嵌套, 直到所有子目录下的变异任务全部完成。

\$(subdir):

```
$(sub_make) -C $@ $(MAKECMDGOALS)
```

命令中 -C 表示进入 \$@ 代表的子目录后执行:

```
$(sub_make) $(MAKECMDGOALS)
```

命令, \$(subdir) 表示在此处把 subdir 变量展开, \$@ 是 makefile 中比较晦涩难懂的符号, 意思是逐个取出目标中单词生成命令。\$(MAKECMDGOALS) 代表命令行输入的最终目标串, 因此,

\$(subdir):

```
$(sub_make) -C $@ $(MAKECMDGOALS)
```

就被展开成:

app:

```
$(sub_make) -C app $(MAKECMDGOALS)
```

djyos:

```
$(sub_make) -C djyos $(MAKECMDGOALS)
```

3、最终目标

debug、run_inram、run_inflash、boot_rom 是我们要生成的最终目标, 他们的生成规则如下:

```
debug \
```

```
run_inram \
```

```
run_inflash \
```

```
boot_rom: rm_obj_list $(subdir)
```

```
$(sub_make) -f make_ld $(MAKECMDGOALS)
```

可以看到, 最终目标依赖 rm_obj_list 目标和 \$(subdir) 目标, 这会引发生成这两个目标。最后一句是把所有 .o 文件连接成可执行文件, -f 指示使用 make_ld 文件替代 makefile, 该文件描述的是连接规则, 参见第 4.4 节。

4.3. 自动生成连接脚本 ld 文件

编译产生的是可重定位的目标文件, 目标文件中不包含地址信息, 需要经过连接后才能生成真正可执行的文件。连接脚本文件就是告诉链接器如何连接目标文件的脚本文件, 不同的编译系统对脚本文件有不同的要求, 一般来说, 脚本文件完成以下功能, ld 文件的详细说明参考 gnu 相关文档。

1、存储器描述, 描写系统中的存储器配置, 比如 44b0 版本的 ld 文件的存储器描述如下:

```
MEMORY
```

```
{
```

```

rom (rx) : org = 0,          len = 2M
ram (wx) : org = 0x0c000000, len = 8M
}

```

表示硬件配置了两块存储器，第一块是只读和可执行的，起始地址是 0，长度是 2M，对应的是 norflash；第二块是可读写和执行的，起始地址为 0xc000000，长度 8M，对应 sdram。

2、存储器分配，即程序中的代码、变量、栈等在存储器中如何分布。

3、输出可供连接器识别的符号，比如 djyos 的存储器管理模块就需要 `cfg_heap_top` 和 `cfg_heap_bottom` 两个符号来确定堆的起止地址。

既然 ld 文件要指挥链接器把代码和数据放到存储器中合适的位置，而 ld 文件又是自动生成的，那具体某一个.c 或.s 文件，谁告诉“ld 文件生成器”，我该放在哪里呢？对了，这是必须由程序员决定的，在子 makefile 中，用这几个变量：`boot_rom`、`rom_init`、`preload`、`sysload`、`critical` 表示具体文件的连接属性的。所有子 makefile 共同生成 `ld_obj_list` 文件中，该文件中用以下格式生成几个变量，生成方法和变量含义参见于 makefile 的说明。

```

rom_init_obj +=
boot_rom_obj +=
critical_obj +=/cygdrive/g/siyf/app/critical/gpio.o
preload_obj +=
sysload_obj +=

```

要生成 ld 文件，有了 `ld_obj_list`，就万事俱备，只欠东风了。`*.mld` 文件隆重登场了，该文件实际上是 makefile 文件的一部分，由 `make_ld` 文件用：

```

ifneq ($(target),)
include $(target).mld
endif

```

包含进来的，5 个最终目标 `debug`、`run_inram`、`run_inflash`、`boot_rom` 对应 5 个 `*.mld` 文件，该文件对应 5 个用于描述 5 个最终目标的连接脚本文件。`*.mld` 文件非常简单，就是用 1 条 shell 内置函数 `printf` 生成 `*.ld` 文件，并把输出转向保存到 `*.ld` 文件中。

4.4. 连接生成可执行文件

在主控 makefile 中，生成最终目标的规则为：

```

$(target): rm_obj_list $(subdir)
$(sub_make) -f make_ld $(MAKECMDGOALS)

```

在完成 `rm_obj_list` 和 `$(subdir)` 两个目标之后，所有源文件都已经被编译成.o 文件，最后执行：

```

$(sub_make) -f make_ld $(MAKECMDGOALS)

```

这一步把.o 文件连接成可执行文件，make 的规则是从 `make_ld` 文件读入的。

好不容易，终于轮到 `make_ld` 上场了，这个文件并不复杂，这里列出它的全部内容：

```

include ld_obj_list
ifneq ($(target),)
include $(target).mld
endif

debug :$(target).ld
$(CC) $(LDFLAGS) $(preload_obj) $(critical_obj) $(sysload_obj) -o $(target).elf -T$(target).ld
$(objcopy) $(target).elf $(target).bin
$(size) $(target).elf
rm -f ld_obj_list
boot_rom:$(target).ld
$(CC) $(LDFLAGS) $(boot_rom_obj) -o $(target).elf -T$(target).ld

```

```

$(objcopy) $(target).elf $(target).bin
$(size) $(target).elf
rm -f ld_obj_list
run_inram:$(target).ld
$(CC) $(LDFLAGS) $(rom_init_obj) $(preload_obj) $(critical_obj) $(sysload_obj) -o $(target).elf
-T$(target).ld
$(objcopy) $(target).elf $(target).bin
$(size) $(target).elf
rm -f ld_obj_list
run_inflash:$(target).ld
$(CC) $(LDFLAGS) $(rom_init_obj) $(preload_obj) $(critical_obj) $(sysload_obj) -o $(target).elf
-T$(target).ld
$(objcopy) $(target).elf $(target).bin
$(size) $(target).elf
rm -f ld_obj_list

```

有人要问了，既然主控 makefile 只有几十行，加上 make_ld 文件总共不到 120 行，为什么一定要使用独立的 make_ld 文件，用

```
$(sub_make) -f make_ld $(MAKECMDGOALS)
```

语句完成连接，而不是直接在该语句的位置用 make_ld 中的：

```
$(CC) $(LDFLAGS).....
```

语句替代来实现连接呢？其实这就是 djyos 的 makefile 的机巧所在。makefile 是一个描述性的脚本文件，除了一个目标后连续出现的命令是按顺序执行的外，其他的一切，比如变量，include 等，并无先后顺序，比如如下左右两组语句是等价的。

```
a = b                c = $(a)
```

```
c = $(a)            a = b
```

前面提到，ld_obj_list 文件是在执行

```
$(subdir):
```

```
$(sub_make) -C $@ $(MAKECMDGOALS)
```

时自动生成的，我们就要确保实际执行 include ld_obj_list 必须在生成\$(subdir)之后，而把 makefile 和 make_ld 写在同一个文件中的话，显然不能按照这种先后顺序执行，include ld_obj_list 将读入一个空文件。

而现在的做法，最终目标依赖\$(subdir)，可以确保在完成\$(subdir)目标之后，才执行：

```
$(sub_make) -f make_ld $(MAKECMDGOALS)
```

把 include ld_obj_list 写在 make_ld 中，可以确保 include 操作在编译子目录中的源文件后。

5. 子 makefile

所有子 makefile 文件中，第 10 行开始是雷同的，本想在工程根目录下用 mksubinc 文件描述雷同部分，然后在子 makefile 中用：

```
include $(topdir)/mksubinc
```

语句包含进来，但这样 make 时会报错：

```
makefile:11: /cygdrive/g/siyf/mksubinc: No such file or directory
```

经多方努力，暂无解决办法，只好先将就之。

5.1. 源文件和下级目录描述

子 makefile 中用以下几个变量，列出本目录下的子目录和汇编或 C 源文件。要准确理解 currentdir 和 subdir 外的其他变量的含义，需参考《都江堰操作系统与嵌入式系统设计》一书的第 4.7 章

```
currentdir = $(shell pwd)
```

```
subdir = app djyos
```

```
boot_rom =
rom_init =
preload =
sysload =
critical =
```

各变量解释如下：

1、currentdir

顾名思义，表示 makefile 文件所在的路径。

2、subdir

本变量包含本目录下所有包含或可能包含源文件的目录串，在工程根目录下，有 app 和 djyos 两个包含代码的子目录，而 include 目录只包含头文件，不算在内。

3、boot_rom

为用硬件仿真器调试准备的，用于启动 cpu 的代码。如果你要用硬件仿真器调试代码，就必须先用：

```
g:\siyf>make boot_rom
```

命令编译产生 boot_rom.bin 文件，并把这个文件用 jtag 工具烧录到 0 地址的 flash 上。在 ARM 版本上，只有 port_rominit 目录下的 initcpu.s 和 int.s 有 boot_rom 属性。

4、rom_init

rom 初始化部分代码，这些代码在复位后在 rom 中执行，除非你打算利用 MMU 重映射 ram 到 0 地址，否则这些代码无需 copy 到 ram 中。

5、preload

需要预加载的代码，这些代码包括：预加载器、中断管理模块、应用程序的安全钩子代码，这些代码由 rom_init 中的预加载器 copy 到 ram 中，然后把控制权交给预加载器。

6、sysload

操作系统内核代码和应用程序代码（除安全钩子部分）。

7、critical

应用程序提供的安全钩子代码，本部分其实属于 preload 代码，只是为了逻辑上更加清晰，用独立的变量表示。

5.2. 生成中间变量

子 makefile 中用以下代码：

```
ifneq ($(boot_rom),)
boot_rom_asm=$(filter %.s,$(boot_rom))
boot_rom_asmobj=$(patsubst %.s,%o,$(boot_rom_asm))
boot_rom_c=$(filter %.c,$(boot_rom))
boot_rom_cobj=$(patsubst %.c,%o,$(boot_rom_c))
boot_rom_obj=$(patsubst %.o,$(currentdir)/%.o,$(boot_rom_asmobj) $(boot_rom_cobj))
endif
```

推导出 boot_rom_asmobj、boot_rom_cobj、boot_rom_obj 三个中间目标变量，boot_rom_asmobj 是 boot_rom 中的汇编源文件生成的.o 文件，boot_rom_cobj 是 boot_rom 中的 c 源文件生成的.o 文件，boot_rom_obj 是 boot_rom 中的所有源文件生成的.o 文件。这样的代码共 5 组，分别对应 boot_rom、rom_init、preload、sysload、critical 这 5 个变量。

5.3. 填写 ld_obj_list 文件

这一步是 makefile 能自动生成连接脚本文件的关键，使用下列代码：

```
.PHONY: output_obj_list
output_obj_list:
    @printf "rom_init_obj +=$(rom_init_obj)\n\
boot_rom_obj +=$(boot_rom_obj)\n\
critical_obj +=$(critical_obj)\n\
preload_obj +=$(preload_obj)\n\
sysload_obj +=$(sysload_obj)\n" >> $(topdir)/ld_obj_list
```

把上一步生成的 boot_rom_obj、rom_init_obj、critical_obj、preload_obj、sysload_obj 这几个变量的值追加到根目录下的 ld_obj_list 文件中。由于是往文件追加内容，所以每次重新编译之前，必须把 ld_obj_list 文件删除。

5.4. 编译

子 makefile 的最终目标，就是把本目录下的源文件编译成.o 文件，以及调用子目录下的 makefile 编译下级目录的源文件。以 debug 目标为例，规则如下：

```
debug :$(subdir) $(preload_asmobj) $(preload_cobj) $(critical_asmobj) \
        $(critical_cobj) $(sysload_asmobj) $(sysload_cobj) output_obj_list
```

debug 的第一个依赖目标是\$(subdir)，这会引发执行下列规则：

```
.PHONY:$(subdir)
$(subdir) :
    $(sub_make) -C $@ $(MAKECMDGOALS)
```

进入所有下级目录执行 make。

然后依赖 preload_asmobj 等目标，将是 make 寻找生成 preload_asmobj 的规则并执行之：

```
$(preload_asmobj):%.o:%.S
    $(CC) $(ASFLAGS) $(incdir) $< -o $@
```

特别提示，%.S 中的 S 必须大写，否则汇编源文件中的条件编译将不起作用且不警告。

最后一个依赖目标是 output_obj_list，将使前一节所述的，填写 ld_obj_list 的操作得以执行。