CS6257 软件工程文档

Version : V1.1

Prepared by : Zhangh

Release Date : 2011-11-30

前言

为了更好的让用户了解 CS6257 软件的结构与使用方法，我们提供了一个简易的 demo 工程供参考。下面将以此 demo 为基础，分以下几个部分分别说明。

1. 工程目录结构说明
2. 代码编辑环境说明
3. 编译器以及编译流程说明
4. 程序编译模式说明
5. 库函数说明
6. 如何添加自己的应用程序
7. 如何在代码中添加中断服务程序
8. 如何编译程序
9. elf 文件、dump 文件以及 bin 文件介绍

一．工程目录结构说明

将压缩的 demo 工程解压后，可以看到下面的目录树结构，表一中将对其做简要说明。

```
.
|-- Makefile
|-- Makefile.eflash
|-- Makefile.ram
|-- bin_tool
|     |-- asmcheck
|     |-- bootmfg
|     |-- cc1
|     |-- cmparse
|     |-- codesize
```

```
|      |-- cpp
|      |-- htmlstrip
|      |-- linkscripts
|      |-- merge_romprof
|      |-- mipsx-elf-ar
|      |-- mipsx-elf-gcc
|      |-- mipsx-elf-nm
|      |-- mipsx5-elf-ax
|      |-- mipsx5-elf-gdb
|      |-- mipsx5-elf-gdb.start
|      |-- mipsx5-elf-gdb.starte
|      |-- mipsx5-elf-nld
|      |-- mipsx5-elf-objcopy
|      |-- mipsx5-elf-objdump
|      |-- pdump
|      |-- rom
|      |-- rom.boot
|      |-- rom.loaderapp
|      |-- rom.profile
|      |-- tpi
|      |-- tpir
|      `-- zipsmall
|-- gdb
|      |-- b
|      |-- burn
|      |-- burner
|      `-- gdb
|-- src
|      |-- common
|      |-- kodiac_6257.h
|      |-- main.c
|      `-- uart
|-- src_files.mk
`-- start
        |-- baseregs.h
        |-- dstart.c
        `-- tstart.c
```

表 一

目录/文件名	说　　明
Makefile	工程的默认 Makefile 文件
Makefile.eflash	以 linear rom 方式编译程序用到的 Makefile 文件
Makefile.ram	以 ram only 方式编译程序用到的 Makefile 文件
bin_tool	该目录下存放 CS6257 Linux 交叉编译器相关工具
gdb	该目录下存放 CS6257 GDB 调试以及程序烧录相关工具。 b　　快速烧录脚本文件，用户可根据要烧录的文件自行修改 burn　　烧录命令脚本文件。 burner　　烧录过程中 load 到 CS6257 中运行的 elf 文件。 gdb　　CS6257 用到的 gdb 调试程序。
src_files.mk	该文件被 Makefile 调用，用户需要将要编译的所有源文件都添加到该文件中。
src	用户应用程序源文件所在目录。用户将要需要编译的源文件放在此目录下，支持多级子目录。
start	存放系统引导与初始化相关源代码。用户无需修改。

二．代码编辑环境介绍

需要编译的源代码，存放在 linux host 上，可在本地或者通过 ssh 远程登录 linux Host 通过 vim 等 linux 编辑工具直接编辑；如果不习惯通过 vim 等 linux 环境里面的工具进行代码编辑，也可以通过在 linux host 配置并开启 samba 服务，在局域网其他 windows 主机上使用 windows 上的编辑器进行代码编辑，如 UltraEdit，Source Insight 等编辑器。

三． 交叉编译器及编译流程说明

1. 交叉编译器说明

CS6257 使用的是 MIPSX5 内核，编译 CS6257 的代码，需要使用到 mipsx5 交叉编译工具链（cross compilation tool chain）,所有相关工具位于 bin_tools 目录下，由 Makefile 文件调用执行。以下是几个主要工具的说明。

mipsx-elf-gcc：mipsx 平台的 GNU C 编译器。

mipsx-elf-nm： 列出目标 elf 文件中的符号。

mipsx-elf-ar：建立、修改、提取归档文件。归档文件是包含多个文件内容的一个大文件，其结构保证了可以恢复原始文件内容。

mipsx5-elf-ax ：主要用来编译 GNU C 编译器 gcc 输出的汇编文件，产生的目标文件由连接器 ld 连接。

mipsx5-elf-nld：链接器，把一些目标和档案文件结合在一起，重新定位数据，并连接符号引用。通常，建立一个新编译程序的最后一步就是调用 nld。

mipsx5-elf-objcopy：把一种目标文件中的内容复制到另外一种目标文件中

mipsx5-elf-objdump：显示一个或多个目标文件的信息，有多种选项来控制显示信息。

codesize：列出目标文件每一段的大小以及总体的大小。

2. 编译流程介绍

在工程根目录敲 make 命令，编译器会将 src_file.mk 里面 SRCFILES 指定的 C 文件编译成.s 文件，之后由汇编器将指定的.s 文件转换成.o 文件，之后通过链接器 linker 链接成 elf 文件，最后通过工程自带的 rom 工具将 elf 加上 bootloader 部分转换成 bin 文件，即最后生成的 r0 文件。

四．程序编译模式介绍

CS6257 软件编译支持 2 种模式，分别为 ram only 和 linear rom，对应的 Makefile 文件分别为 Makefile.ram 和 Makefile.eflash。使用哪种编译模式，需要将对应的 makefile 文件替换掉工程根目录下的文件 Makefile。

使用 ram only 模式编译，生成 image 的 text 段位于 sram 中。图一为编译后生成的 dump 文件里面记录的数据段信息，可以看出 text 段起始地址为 ox5d0。如果用户程序 text 段和程序消耗的 sram 空间小于 64KB（CS6257 片内 sram 总大小为 64k），调试时可直接将生成的 elf 文件 load 到 sram 中，不需要每次调试都烧录。

使用 linear rom 模式编译，生成 image 的 text 段位于片内 flash 中。图二

为编译生成的 dump 文件里面记录的数据段信息，可以看出 text 段起始地址为 ox2014a8，位于 flash 地址空间。每次调试程序前，需要将生成的 image 烧录到 flash。

```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .data         0000022c  00000200  00000200  000007b4  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  1 .bss          0000000c  0000042c  0000042c  00000000  2**2
                  ALLOC
  2 .rodata       000000a8  00000438  00000438  000009e0  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .lrodata      000000f0  000004e0  000004e0  00000a88  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .text         000014f0  000005d0  000005d0  00000b78  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  5 .stab         00002910  00000000  00000000  00002068  2**2
                  CONTENTS, READONLY, DEBUGGING

                  CONTENTS, READONLY, DEBUGGING
```

图一

```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .data         0000022c  00000200  00000200  00000820  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  1 .bss          0000000c  0000042c  0000042c  00000000  2**2
                  ALLOC
  2 .text         000000d8  00000438  00000438  00000a4c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  3 .rodata       000000a8  00201400  00201400  00000b24  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .lrodata      000000f0  002014a8  002014a8  00000bcc  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .romtext      00001418  00201598  00201598  00000cbc  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  6 .stab         00002a0c  00000000  00000000  000020d4  2**2
                  CONTENTS, READONLY, DEBUGGING
  7 .stabstr      000015b8  00000000  00000000  00004ae0  2**0
                  CONTENTS, READONLY, DEBUGGING
```

图二

五．库函数说明

工程中 common/utility.c 中自带若干字符串操作库函数，可被应用程序直接调用。详细内容如下：

1．int memcmp(OCTET* s1, OCTET* s2, DWORD size)

【功能】字符串比较

【参数】s1：源字符串 1；s2：源字符串 2；size：要比较的字符串长度

【返回值】0：两个字符串一致；非 0：字符串 s1 与 s2 首个不相同字符之差。

2．int memcpy(char *s1, char *s2, int size);

【功能】从字符串 s2 拷贝长度为 size 字节的数据到 s1

【参数】s1：目标字符串；s2：源字符串；size：要拷贝的字节数

【返回值】返回 0

3．char* strpbrk(char *s, char *breakat);

【功能】依次检验字符串 s 中的字符，只要字符串 breakat 中有任何一个字符和被检验字符串中的字符匹配，则停止检验，并返回该字符位置，空字符 NULL 不包括在内。

【参数】s：被搜索字符串；breakat：搜索关键词

【返回值】NULL：在 s 中没有找到匹配的字符串；非 0：匹配字符的位置。

4. char* strchr(char *s, char charwanted);

【功能】查找字符串 s 中首次出现字符 charwanted 的位置

【参数】s：被检索字符串；charwanted：目标字符

【返回值】0：没有找到匹配字符；非 0：匹配字符在 s 中的位置

5. int strspn(char *s, char *accept);

【功能】回字符串中第一个不在指定字符串中出现的字符下标

【参数】s：被搜索字符串；accept：匹配模式字符串

【返回值】返回字符串 s 开头连续包含字符串 accept 内的字符数目。

6. int strcspn(char *s, char *reject);

【功能】顺序在字符串 s 中搜寻与 reject 中字符的第一个相同字符，返回这个字符在 S 中第一次出现的位置。

【参数】s：被搜索字符串；reject：匹配模式字符串

【返回值】返回第一个出现的字符在 s 中的下标值，亦即在 s 中出现而 reject 中没有出现的子串的长度。

7. int strlen(char *scan);

【功能】测试字符串长度

【参数】scan：被检测字符串

【返回值】字符串长度

8. char* strcpy(char *s1, char* s2);

【功能】字符串拷贝

【参数】s1：目标字符串；s2：源字符串

【返回值】目标字符串地址

六．如何添加自己的应用程序

用户应用程序源码，全部放在 src 目录下，入口函数名为 main()。支持多级目录。用户可根据自己的需求进行组织。

本 demo 工程，主要实现了简单的串口打印和 timer 中断处理，src 目录结构如下：

```
.
|-- common
|     |-- utility.c        （几个通用函数的定义）
|     `-- utility.h
|-- kodiac_6257.h    （寄存器定义头文件）
|-- main.c                （主程序）
`-- uart
      |-- uart.c          （串口驱动以及串口打印 printf 格式化函数的实现）
      `-- uart.h
```

用户只需要根据当前的文件结构，将需要编译的 c 文件加入到根目录下的 src_files.mk 中即可。下面为 demo 中 src_files.mk 内容：

```
SRCFILES := main.c \
                    uart/uart.c\
                    common/utility.c
```

注意：如果是多级目录，需要与 src 下的目录结构保持一致。多行书写用“\”分隔。

七．如何在代码中添加中断服务程序

CS6257 不支持中断嵌套，中断触发后，硬件自动禁用中断，然后程序跳转到全局中断服务程序入口地址（一般为地址 0），在全局中断服务程序中，通过查询寄存器 IRQ_VECT 确定触发中断的中断号，然后执行相应的中断服务程序。处理完成后，退出中断服务流程，并开启中断。

下面将结合代码介绍 CS6257 中断机制和相关的处理函数以及如何添加中断服务程序。详细代码请参考 demo 工程 src 下的 main.c。

在开启中断前需要完成中断初始化配置。

```
//------------install IRQ ---------------------------
setupirq(install_irq);
```

在 setupirq 中，首先将全局中断服务程序入口地址设为 0。然后将”install_irq”开始的 8 条指令复制到全局中断服务程序的入口地址 0。

```
/* ********* Interrupt installer ********** */
void setupirq(void (*irqcode))
{
  int *a, i, *b;

  asm("addi r0,#0xf,r1");
  asm("movtos r1,psw");
  kodiac[IRQ_BASE] = 0;
  (long)irqcode = ((long)irqcode) << 2;
  for (a = (int *)0, b = irqcode,  i = 0; i < 8; i++)
    *a++ = *b++;
}
```

install_irq 开始的 8 条指令

```
/* ******** Routine stored at location 0 ******** */
asm(".noreorg");
asm(".globl    _install_irq");
asm("\n_install_irq:");
asm(".noreorg");

asm("nop");
asm("jmpi    r24,#_dmaint");
asm("nop");
asm("nop");
asm("nop");
asm("nop");
asm("nop");
asm("nop");

asm(".end");
asm(".reorgon");
```

中断触发后，会跳转到 0 地址执行以上指令。跳转到函数 dmaint。

函数 dmaint 中保存 r31 后，跳转到函数_dmaint。

```
asm(".globl _dmaint\n_dmaint:\n");
asm("addi r29,#-4,r29");
asm("st 0[r29],r31");
asm("jspci r24,#__dmaint");
asm("nop");
asm("nop");
asm("ld 0[r29],r31");
asm("addi r29,#4,r29");
asm("\n.noreorg");
asm("jpcrs");
asm("nop");
asm("nop");
asm(".reorgon");
```

在函数_dmaint 中首先保存现场，将通用寄存器 r0~r31 压入堆栈，然后读寄存器 IRQ_VECT，查询触发当前中断的中断源，根据中断源类型调用对应的中断服务程序。然后恢复现场，退出函数。

注：以下代码中 kodiac[IRQ_VECT]，是特别定义的一种寄存器访问方式。其中 kodiac 是一个值为 r27 的指针，IRQ_VECT 为中断向量寄存器偏移地址的宏定义，内容如下：

#define IRQ_VECT　6

register volatile int *kodiac asm("r27");

r27 为内部寄存器基地址，在程序的初始化代码中，将其固定为 0x1f0000。

所以 kodiac[IRQ_VECT]等价与地址（0x1f0000+6*4）所存储的内容，也就是中断向量寄存器地址（地址为 0x1f0018）的内容

```
/* ---------- Interrupt Service Routine ------------ */

  irq_vect = kodiac[IRQ_VECT];
  if (irq_vect != 0){
        switch (irq_vect - 1){
                case 0: /* Timer 1*/
                        timer1_isr();
                        break;
                case 1: /* Timer 2 */
                        break;
                case 17: /* AD */
                        break;
                case 21: /* Timer 3 */
                        break;
                case 23: /* SPI Flash */
                        break;
                case 24: /* on chip flash */
                        break;
                case 25: /* pwm interrupt */
                        break;
                case 28: /* watch dog */
                        break;
                default:
                        break;
        }
        irq_vect = kodiac[IRQ_VECT];
  }
```

以上部分为中断处理的基本流程。

用户在添加中断处理时，只需要在 install IRQ 后， 根据中断类型，做初始化配置，并将寄存器 IRQ_MASK 中断对应的位写为 1，来打开中断。每个中断对应的中断号请参看 CS6225 datasheet IRQ_MASK 寄存器。

以 demo 工程中添加 timer 1 中断为例，代码如下：

1. 在 main()中初始化中断。

```
//------------install IRQ ------------------
setupirq(install_irq);


//------------ Timer1 interrupt setting ----

kodiac[TIMER1CMP] = 0x17f7840; // 1s
kodiac[TIMER1CNT] = 0;
kodiac[IRQ_MASK] |= (1<<0); // timer1
```

2.在函数_dmaint（）中添加 timer1 的服务函数，对应的中断号为 0，中断服务函数为 timer1_isr()

```
/* ---------- Interrupt Service Routine ------------ */

  irq_vect = kodiac[IRQ_VECT];
  if (irq_vect != 0){
        switch (irq_vect - 1){
                case 0: /* Timer 1*/
                        timer1_isr();
                        break;
                case 1: /* Timer 2 */
                        break;
                case 17: /* AD */
                        break;
                case 21: /* Timer 3 */
                        break;
                case 23: /* SPI Flash */
                        break;
```

3. 定义函数 timer1_isr()

```
//------ timer1 interrupt service routing------
void timer1_isr(void)
{
        kodiac[TIMERSTAT] &= ~(1<<1);
        kodiac[TIMER1CMP] = 0x17f7840;
        kodiac[TIMER1CNT] = 0;
        timerIntTimes++;
}
```

八．如何编译程序

用户根据选择的编译模式，将根目录下的 Makefile.ram 或者 Makefile.eflash 替换掉 Makefile。例如，我们要编译 ram only 的程序，操作如下：

```
[root@localhost 6257demo]# cp Makefile.ram Makefile
```

然后在工程根目录下敲 make（或者 make –s：略去编译过程中的打印信息）

```
[root@localhost 6257demo]# make
```

编译完成后，会在根目录下生成 bin 目录，我们需要用到其中 3 个文件，分别是：

eflash : elf 文件，用于程序 debug。

eflash.r0 : bin 文件，此文件可烧录到 flash 中运行。

eflash.dmp ：此文件记录着程序编译后 image 中各个数据段的起始地址和大小，全局变量和函数的地址，c 程序对应的 assembly code 等等信息，可作为调试时分析问题的辅助参考手段。

然后根据调试或者烧录程序的需求，将 eflash、eflash.r0 复制到工程目录下的 gdb 目录下备用。

```
[root@localhost 6257demo]# cp bin/eflash bin/eflash.r0 gdb/
```

如果需要清理编译生成的中间文件，可在工程根目录下敲

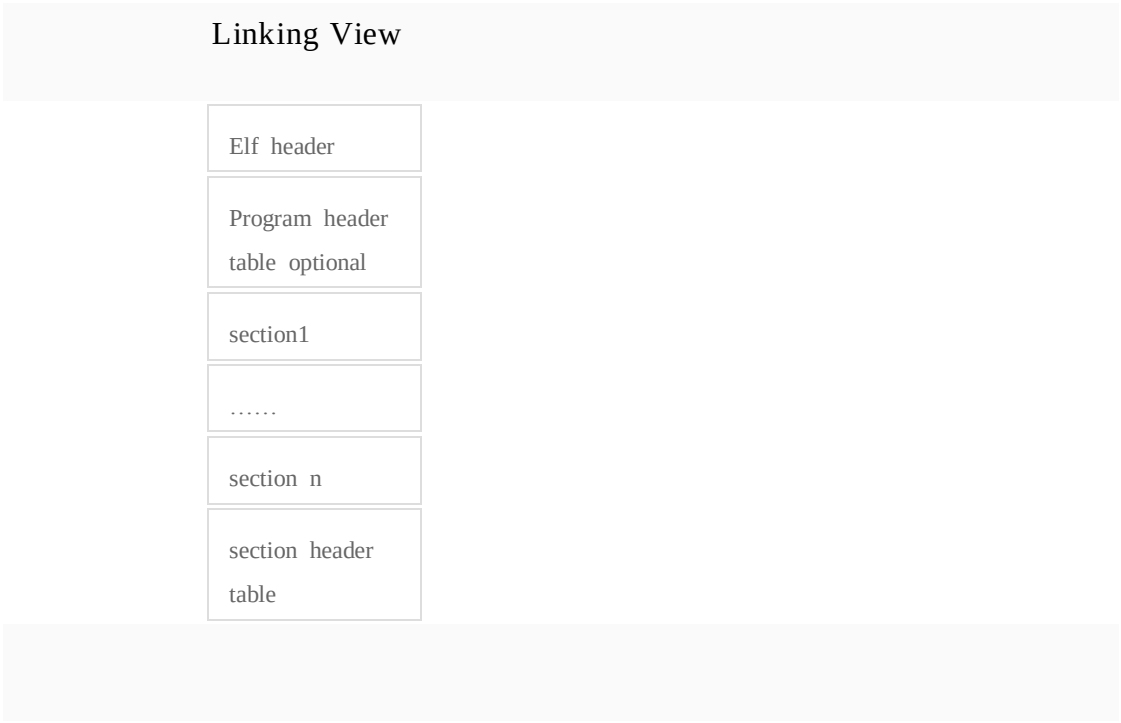
make cleaneverything

```
[root@localhost 6257demo]# make cleaneverything
rm -f -r .x  bin
[root@localhost 6257demo]#
```

九．elf 文件、dump 文件以及 bin 文件介绍

1. ELF 文件说明

ELF = Executable and Linkable Format，可执行连接格式，是 UNIX 系统实验室（USL）作为应用程序二进制接口（Application Binary Interface，ABI）而开发和发布的。工具接口标准委员会(TIS)选择了正在发展中的 ELF 标准作为工作在 32 位 INTEL 体系上不同操作系统之间可移植的二进制文件格式。



一个 ELF 头在文件的开始，保存了路线图(road map)，描述了该文件的组织情况。sections 保存着 object 文件的信息，从连接角度看：包括指令，数据，符号表，重定位信息等等。

假如一个程序头表（program header table）存在，那么它告诉系统如何来创建一个进程的内存映象。被用来建立进程映象(执行一个程序)的文件必须要有一个程序头表（program header table)；可重定位文件不需要这个头表。

一个 section 头表（section header table）包含了描述文件 sections 的信息。每个 section 在这个表中有一个入口；每个入口给出了该 section 的名字，大小，等等信息。在链接过程中的文件必须有一个 section 头表；其他 object 文件可要可不要这个 section 头表。　　注意：虽然图显示出程序头表立刻出现在一个 ELF 头后，section 头表跟着其他 section 部分出现，事实是的文件是可以不同的。sections 没有特别的顺序。只有 ELF 头（elf header）是在文件的固定位置。

2. dump 文件说明

在 makefile 中，通过调用 mipsx5-elf-objdump，在 elf 文件 eflash 基础上生成 dump 文件 elfash.dmp。

该文件记录着 elf 文件各个数据段和代码段的大小、起始地址；符号表；以及 text 段的 disassembly code。此文件可作为辅助调试手段之一。

```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .data         0000022c  00000200  00000200  00000820  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  1 .bss          0000000c  0000042c  0000042c  00000000  2**2
                  ALLOC
  2 .text         000000d8  00000438  00000438  00000a4c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  3 .rodata       000000a8  00201400  00201400  00000b24  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .lrodata      000000f0  002014a8  002014a8  00000bcc  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .romtext      00001418  00201598  00201598  00000cbc  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  6 .stab         00002a0c  00000000  00000000  000020d4  2**2
                  CONTENTS, READONLY, DEBUGGING
  7 .stabstr      000015b8  00000000  00000000  00004ae0  2**0
                  CONTENTS, READONLY, DEBUGGING
```

例如上图是 dump 文件的起始位置会有编译出来的 elf 文件各个 section 的位置 VMA 和 LMA 表示段的起始地址，Size 表示段的大小。Data 段表示全局

变量，放在 ram 里面的段，bss 段表示没有初始化的全局变量段，也是放在 ram 内，text 表示放在 ram 里面运行的程序段，romtext 表示放在 flash 上运行的程序段，rodata 和 lrodata 是只读数据段，放在 flash 里，stab 和 stabstr 是调试信息段

3. BIN 文件

编译生成 elf 文件后，makefile 调用 bin_tool 目录下的 rom 工具，解析并提取 elf 文件中各个段，并插入 loader 程序 rom.boot，生成可执行映像文件 eflash.r0。将该文件烧录到 flash 中，可自动运行。