Sistemas Embebidos I
Programas para ROM – parte 2

O espaço de memória de um sistema embebido é ocupado por memória do tipo RAM (SRAM ou SDRAM) e por memória do tipo ROM (FLASH ou EEPROM). A memória FLASH é usada para armazenar o programa e a RAM é usada para suporte à execução do programa (variáveis e *stack).*

O código da aplicação pode ser executado directamente em ROM ou em RAM.

No segundo caso, o programa é previamente copiado da ROM para a RAM e depois executado. O programa pode ser armazenado em diversos formatos, como por exemplo: binário puro, directamente interpretável pelo processador; compactado, para reduzir a necessidade de espaço; ou em formatos normalizados como ELF, A.OUT, EXE, HexIntel, Srecord, etc. O programa de carregamento é responsável pela interpretação do formato de armazenamento e por colocar o programa acessível ao processador para execução em formato binário natural.

No estágio inicial da sua actividade o processador executa um programa gravado em memória ROM e acessível no "endereço de *reset*". Este programa terá as instruções necessárias para estabelecer um ambiente normal de trabalho: acesso aos periféricos e acesso à memória. A forma como o processamento evolui é muito variável. Em pequenos sistemas embebidos este programa pode ser a aplicação final. Noutros casos, este programa é apenas o primeiro de uma sucessão de carregamentos (vários estágios de boot → sistema operativo → aplicação).

Composição de um programa

O executável de um programa é composto basicamente por código e dados. A zona de dados engloba as constantes, as variáveis iniciadas, as variáveis não iniciadas e a pilha do processador. Os componentes do programa, pertencentes a cada uma das categorias referidas, são distribuídos por grupos segundo a sua natureza, dando origem a pelo menos cinco grupos: um para o código e quatro para cada uma das categorias de dados. Estes grupos designam-se por secções.
A criação de secções de natureza diferente permite um manuseamento adequando das partes do programa no armazenamento e na preparação da execução.
No armazenamento do programa, guarda-se, por cada secção, propriedades como: a dimensão, o endereço final, o alinhamento, etc. Para as secções com código, com constantes ou com valor inicial de variáveis é necessário guardar também o conteúdo.
Na execução do programa as secções de dados, excepto as constantes, têm que ser alojadas em memória RAM, o código e as constantes podem ser alojadas em RAM ou em ROM.
Para um programa em C/C++ o compilador GNU produz secções com as designações seguintes:

- **.text** - para o código;
- **.data** - para as variáveis com valor inicial definido;
- **.bss** - para as variáveis não iniciadas;
- **.rodata** - para as constantes;
- **.stack** - para a pilha do programa.

Efectivamente, podem existir mais secções, mas enquadram-se nestas cinco categorias.

Preparação para a execução

A preparação para a execução de um programa em C inclui, entre outras, as seguintes operações: preparar as zonas de memória; preenchê-las com o conteúdo das secções do programa; iniciar registos do processador; passar os parâmetros à função **main**.

A preparação pode ser feita por um programa auxiliar (*debugger* ou *bootloader*), pelo próprio programa ou ser repartida por ambos.

Execução de um programa na memória RAM do target, usando *debugger* e ligação JTAG.

A memória RAM encontra-se disponível, pelo que não requer preparação. O *debugger* envia para a RAM, via JTAG, o conteúdo das secções **.text .rodata** e **.data**. As secções .**bss** e **.stack** são tratadas pelo próprio programa no módulo de arranque (ver **cstart_ram.s).**

Execução directa de um programa gravado na ROM

Neste caso é necessário gravar previamente em ROM as secções **.text .rodata** e **.data**. A secção **.text** deve ser colocada numa posição da ROM que, após a activação do sinal *reset* (ligação da energia), seja directamente acessível pelo processador. O programa começa por fazer as iniciações de hardware, depois preenche as variáveis iniciadas, copiando a secção **.data** da ROM para a RAM (ver **cstart_rom.s**) e prossegue como no caso anterior.

Execução em RAM de um programa gravado na ROM

Nalgumas situações pode ser conveniente executar o programa em RAM (por ser mais rápida, mais abundante, etc). Neste caso, é necessário copiar também as secções **.text** e **.rodata** da ROM para a RAM.

Preparação do programa

A fase de preparação para a execução de um programa em C está a cargo do módulo de programa designado por "**modulo de arranque**" (cstart). Este módulo deve garantir que, ao se invocar a função **main**, estão criados os pressupostos de execução de um programa em C. Após a actuação do módulo de arranque, que é dependente da situação de instalação, um mesmo programa executa sempre da mesma forma.

Vamos começar por apresentar a preparação para a execução em RAM (cstart_ram.s e ldscript_ram) e em seguida a preparação para a execução em ROM (cstart_rom e ldcript_rom).

Módulo de arranque para execução em RAM (cstart_ram.S)

Neste caso, o programa (secções **.text**, **.rodata** e **.data**) é previamente carregado em RAM. O módulo de arranque completa a preparação colocando a secção **.bss** a zero e iniciando o registo SP. em seguida chama a função **main** com **argc** e **argv** a 0. Se a função **main** retornar é chamada a função **exit** que no caso de um sistema embebido não deve acontecer.

```
/*   Instituto Superior de Engenharia de Lisboa
     Ezequiel Conde, 2009

     cstart to run programs in RAM with JTAG
*/

/*-------------------------------------------------------------------------
     Exception table. Must be mapped to 0x00000000
*/
     .text

     ldr       pc, _reset_handler
     ldr       pc, _undefined_handler
```

```
    ldr       pc, _swi_handler
    ldr       pc, _prefetch_abort_handler
    ldr       pc, _data_abort_handler
    .word     0
/*  ldr       pc, [pc, #-0xFF0] */
    ldr       pc, _irq_handler
    ldr       pc, _fiq_handler

_reset_handler:             .word     reset_handler
_undefined_handler:         .word     undefined_handler
_swi_handler:               .word     swi_handler
_prefetch_abort_handler:    .word     prefetch_abort_handler
_data_abort_handler:        .word     data_abort_handler
_irq_handler:               .word     irq_handler
_fiq_handler:               .word     fiq_handler

/*-------------------------------------------------------------------------
    Default handlers
*/

reset_handler:
    b         _start

undefined_handler:
    mrs       r0, spsr
    msr       cpsr_c, r0
    b         undefined_handler

swi_handler:
    mov  pc, lr

prefetch_abort_handler:
    mrs       r0, spsr
    msr       cpsr_c, r0
    b         prefetch_abort_handler

data_abort_handler:
    mrs       r0, spsr
    msr       cpsr_c, r0
    b         data_abort_handler

fiq_handler:
    mrs       r0, spsr
    msr       cpsr_c, r0
    b         fiq_handler

irq_handler:
    mrs       r0, spsr
    msr       cpsr_c, r0
    b         irq_handler

/*-------------------------------------------------------------------------
    Program entry point
*/
    .global   _start
_start:

/*-------------------------------------------------------------------------
    Inicializar o stack para cada modo
*/
    ldr       r0, =__stack_end__
```

```
    mov       r1, #0xd1     /*   FIQ  */
    msr       CPSR_c, r1
    mov       sp, r0
    sub       r0, r0, #3*4

    mov       r1, #0xd2     /*   IRQ  */
    msr       CPSR_c, r1
    mov       sp, r0
    sub       r0, r0, #16*8*4

    mov       r1, #0xd7     /*   Abort    */
    msr       CPSR_c, r1
    mov       sp, r0
    sub       r0, r0, #1*4

    mov       r1, #0xdb     /*   Undefined */
    msr       CPSR_c, r1
    mov       sp, r0
    sub       r0, r0, #1*4

    mov       r1, #0xd3     /*   Supervisor    */
    msr       CPSR_c, r1
    mov       sp, r0
    sub       r0, r0, #1*4

    mov       r1, #0xdf     /*   System    ; disable interrupts */
    msr       CPSR_c, r1
    mov       sp, r0

/*-------------------------------------------------------------------------
    Inicializar o fp (não é imprescindivel)
*/
    mov       fp, #0

/*-------------------------------------------------------------------------
    Colocar o BSS a zero
*/
    ldr       r1,=__bss_start__
    ldr       r2,=__bss_end__
    mov       r0, #0
    cmp       r1, r2
    bhs       2f
1:
    str       r0, [r1], #4
    cmp       r1, r2
    blo       1b
2:

/*-------------------------------------------------------------------------
*/

    mov       r0, #0        ;    argc = 0
    mov       r1, #0        ;    argv = 0
    bl        main
    bl        exit
```

Script de localização em RAM (ldscript_ram)

A directiva ENTRY serve para indicar o endereço da primeira instrução do programa a ser

executada (*entry point*).

Na directiva MEMORY identifica-se a área de memória RAM disponível – endereço e dimensão.

Sob a directiva SECTIONS enumeram-se as secções de saída, e as secções de entrada que lhes dão origem. As secções de saída irão pertencer ao ficheiro com o programa executável e as secções de entrada pertencem aos ficheiros objecto (*.o). No inicio e no fim de cada secção, são definidas duas *labels* para permitir a manipulação da secção como um bloco. As *labels* têm a forma **__xxx_start__** e **__xxx_end__**. As secções de saída têm também a indicação da zona de memória que vão ocupar em execução **.text { … } > ram**.

```
ENTRY(_start)

MEMORY
{
     ram : ORIGIN = 0x40000000, LENGTH = 0x10000 - 32  /* 32 - usado pelo IAP */
}

SECTIONS
{
     .debug_aranges  0 : { *(.debug_aranges) }
     .debug_pubnames 0 : { *(.debug_pubnames) }
     .debug_info     0 : { *(.debug_info) }
     .debug_abbrev   0 : { *(.debug_abbrev) }
     .debug_line     0 : { *(.debug_line) }
     .debug_frame    0 : { *(.debug_frame) }
     .debug_str      0 : { *(.debug_str) }
     .debug_loc      0 : { *(.debug_loc) }
     .debug_macinfo  0 : { *(.debug_macinfo) }

     /DISCARD/ 0 : { *(.ARM.exidx* .gnu.linkonce.armexidx.*) }

     .text : {
          __text_start__ = ABSOLUTE(.);
          *(.text*) *(.gnu.warning) *(.gnu.linkonce*) *(.init) *(.glue_7)
*(.glue_7t);
          __text_end__ = ABSOLUTE(.);
     } > ram

     .rodata ALIGN(4) : {
          __rodata_start__ = ABSOLUTE(.);
          *(.rodata*);
          . = ALIGN(4);
          __rodata_end__ = ABSOLUTE(.);
     } > ram

     .data ALIGN(4) : {
          __data_start = ABSOLUTE(.);
          *(.data*);
          *(.eh_frame);
          __data_end__ = ABSOLUTE(.) ;
     } > ram

     .bss ALIGN(4) : {
          __bss_start__ = ABSOLUTE(.);
          *(.bss*) *(COMMON);
          __bss_end__ = ABSOLUTE(.);
     } > ram
```

```
    .stack ALIGN(4) : {
         __stack_start__ = ABSOLUTE(.);
         *(.stack);
         . = . + 0x1000;
         __stack_end__ = ABSOLUTE(.);
    } > ram
}
```

Módulo de arranque em ROM (cstart_rom.S)

Este módulo deve ser localizado de modo que a instrução **b _start**, na primeira posição da tabela de excepções, seja visível no endereço zero após *reset* e, como consequência, seja a primeira instrução a ser executada.

Neste módulo começa por se copiar o valor inicial das variáveis, que se encontra gravado em ROM, para a RAM. O restante programa é igual ao cstart para a situação RAM. Recorrendo a compilação condicional poder-se-ia criar um único ficheiro cstart.s para as situações RAM e ROM.

```
/*   Instituto Superior de Engenharia de Lisboa
     Ezequiel Conde, 2010

     cstart to make rommable programs
*/

/*-------------------------------------------------------------------------
     Exception table. Must copied to RAM visible at address 0x00000000
*/
          .text
     b         _start
     ldr       pc, _undefined_handler
     ldr       pc, _swi_handler
     ldr       pc, _prefetch_abort_handler
     ldr       pc, _data_abort_handler
     .word     0xb4405f7e
/*   ldr       pc, [pc, #-0xFF0] */
     ldr       pc, _irq_handler
     ldr       pc, _fiq_handler

_undefined_handler:          .word    undefined_handler
_swi_handler:                .word    swi_handler
_prefetch_abort_handler:     .word    prefetch_abort_handler
_data_abort_handler:         .word    data_abort_handler
_irq_handler:                .word    irq_handler
_fiq_handler:                .word    fiq_handler

undefined_handler:           b    .
swi_handler:                 b    .
prefetch_abort_handler:      b    .
data_abort_handler:          b    .
irq_handler:                 b    .
fiq_handler:                 b    .

/*-------------------------------------------------------------------------
     Program entry point
*/
     .global   _start
_start:
```

```
#if 0
    ldr       r0, =0xe0028000    /* GPIO_BASE */
    mov       r1, #0x10000
    str       r1, [r0, #8]       /* IODIR */
    str       r1, [r0, #4]       /* IOSET */
    str       r1, [r0, #12]      /* IOCLR */
#endif

/*-------------------------------------------------------------------------
    Copiar a secção DATA da ROM para a RAM
*/
    ldr       r1,=__data_start__
    ldr       r2,=__data_end__
    ldr       r3,=__rom_data_start__
    cmp       r1, r2
    bhs       2f
1:
    ldr       r0, [r3], #4
    str       r0, [r1], #4
    cmp       r1, r2
    blo       1b
2:

/*-------------------------------------------------------------------------
    Inicializar o stack para cada modo
*/
    ldr       r0, =__stack_end__

    mov       r1, #0xd1      /*   FIQ  */
    msr       CPSR_c, r1
    mov       sp, r0
    sub       r0, r0, #3*4

    mov       r1, #0xd2      /*   IRQ  */
    msr       CPSR_c, r1
    mov       sp, r0
    sub       r0, r0, #16*8*4

    mov       r1, #0xd7      /*   Abort     */
    msr       CPSR_c, r1
    mov       sp, r0
    sub       r0, r0, #1*4

    mov       r1, #0xdb      /*   Undefined */
    msr       CPSR_c, r1
    mov       sp, r0
    sub       r0, r0, #1*4

    mov       r1, #0xd3      /*   Supervisor    */
    msr       CPSR_c, r1
    mov       sp, r0
    sub       r0, r0, #1*4

    mov       r1, #0xdf      /*   System    ; disable interrupts */
    msr       CPSR_c, r1
    mov       sp, r0

/*-------------------------------------------------------------------------
    Inicializar o fp (não é imprescindivel)
*/
    mov       fp, #0
```

```
/*-------------------------------------------------------------------------
    Colocar o BSS a zero
*/
    ldr       r1, =__bss_start__
    ldr       r2, =__bss_end__
    mov       r0, #0
    cmp       r1, r2
    bhs       2f
1:
    str       r0, [r1], #4
    cmp       r1, r2
    blo       1b
2:

/*-------------------------------------------------------------------------
*/
    mov       r0, #0          ;    argc = 0
    mov       r1, #0          ;    argv = 0
    bl        main
    bl        exit
```

Script de localização em ROM (ldscript_rom)

Neste script foi acrescentada a área de memória designada por **rom**, correspondente à memória FLASH. Começa no endereço 0x00000000 e tem 128 Kbyte de dimensão.

O programa de ligação LD atribui a cada secção dois endereços o VMA (virtual memory address), que corresponde ao endereço de execução e o LMA (loaded memory address), que corresponde ao endereço de carregamento. Em muitas situações estes endereços são iguais, é o caso de programas para RAM. Na produção de programas para ROM tira-se partido desta característica no tratamento da secção **.data**. Usa-se LMA para especificar o endereço onde vai ser gravada na ROM e VMA para especificar o endereço onde vai ser executada.

A localização VMA é indicada pela notação **> ram** ou **> rom** e a localização LMA é indicada pela directiva AT, que neste caso aparece na secção **.data**. A esta secção é atribuído um endereço VMA em memória RAM e um endereço LMA em memória ROM. No caso de não ser indicada a directiva AT o enderreço LMA é igual a VMA. O endereço LMA da secção .data, demarcado pela label __rom_data_start__, coincide com a label __rodata_end__ indicadora de fim da secção .rodata.

```
ENTRY(_start)

MEMORY
{
    ram : ORIGIN = 0x40000000 + 64, LENGTH = 0x10000 - 64 - 32    /* 64 -
tabela de excepções, 32 - usado pelo IAP */
    rom : ORIGIN = 0x00000000, LENGTH = 0x20000
}

SECTIONS
{
    .debug_aranges  0 : { *(.debug_aranges) }
    .debug_pubnames 0 : { *(.debug_pubnames) }
    .debug_info     0 : { *(.debug_info) }
    .debug_abbrev   0 : { *(.debug_abbrev) }
    .debug_line     0 : { *(.debug_line) }
    .debug_frame    0 : { *(.debug_frame) }
    .debug_str      0 : { *(.debug_str) }
```

```
    .debug_loc      0 : { *(.debug_loc) }
    .debug_macinfo  0 : { *(.debug_macinfo) }

    /DISCARD/ 0 : { *(.ARM.exidx* .gnu.linkonce.armexidx.*) }

    .data : AT (__rodata_end__) {
         __data_start__ = ABSOLUTE(.);
         *(.data*);
         . = ALIGN(4);
         __data_end__ = ABSOLUTE(.) ;
    } > ram
    __rom_data_start__ = __rodata_end__;

    .bss ALIGN(4) : {
         __bss_start__ = ABSOLUTE(.);
         *(.bss*) *(COMMON);
         . = ALIGN(4);
         __bss_end__ = ABSOLUTE(.);
    } > ram

    .stack ALIGN(4) : {
         __stack_start__ = ABSOLUTE(.);
         *(.stack);
         . = . + 0x1000;
         __stack_end__ = ABSOLUTE(.);
    } > ram

    .text : {
         __text_start__ = ABSOLUTE(.);
         *(.text*) *(.gnu.warning) *(.gnu.linkonce*) *(.init) *(.glue_7)
*(.glue_7t);
         __text_end__ = ABSOLUTE(.);
    } > rom

    .rodata ALIGN(4) : {
         _rodata_start__ = ABSOLUTE(.);
         *(.rodata*);
         . = ALIGN(4);
         __rodata_end__ = ABSOLUTE(.);
    } > rom
}
```

Gravação do programa em ROM

A informação do programa a gravar na Flash são os conteúdos das secções **.text**, **.rodata** e **.data**.

Após a produção do executável os endereços e outros atributos do programa podem ser visualizados com o utilitário objdump. Por exemplo:

```
$arm-elf-objdump -h main.elf

main.elf:     file format elf32-littlearm

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
...
  7 .data         00000038  40000040  000019fc  00010040  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  8 .bss          00000000  40000078  00001a34  00010078  2**0
                  ALLOC
```

```
  9 .stack        00001000  40000078  40000078  00013fcc  2**0
                  CONTENTS
 10 .text         00001628  00000000  00000000  00008000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 11 .rodata       000003d4  00001628  00001628  00009628  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
...
```

As secções **.bss** e **.stack** têm endereços VMA e LMA iguais e estão localizadas em RAM. As secções **.text** e **.rodata** também têm endereços VMA e LMA iguais e estão localizadas em ROM. A secção **.data** tem endereços VMA em RAM onde irá ser usada pelo programa e endereços LMA em ROM onde irá ser gravada, juntamente com **.text** e com **.rodata**.

O processo de gravação na FLASH pode recolher essa informação do ficheiro executável, em formato **elf**, ou pode ser necessário converter para outro formato. O formato mais comum é o formato binário, em que se produz um ficheiro cujo conteúdo é a imagem exacta daquilo que irá ser o conteúdo na ROM. O dados são posicionadas no ficheiro de imagem de acordo com os endereço LMA, que neste caso formarão um bloco contínuo contendo as secções **.text, .rodata e .data.**

O ficheiro binário pode ser gerado pelo utilitário objcopy a partir do executável em formato **elf**.

```
$ arm-elf-objcopy -Obinary main.elf main.bin
```

Makefile

Apresenta-se em seguida um ficheiro makefile para geração, em alternativa, de programas para execução dom debugger, em RAM, ou para execução em ROM.

Para a geração do programa para execução em RAM fazer:

```
$ make
```

Para a geração do programa para execução em ROM fazer:

```
$ make ROM=1 -B
```

A opção **-B** provoca a geração de todos os *targets*. É necessário porque as opções de compilação podem ser diferentes. Por exemplo o código ROM é optimizado com a opção -Os.

O argumento **ROM=1** corresponde à definição do símbolo **ROM** e que é usado para activar a directiva **ifdef ROM**.

```
HOME = ../..

LIBGCC = `arm-eabi-gcc --print-libgcc-file-name`

VPATH = $(HOME)/mylib:

OBJECTS = main.o lcd.o lcd_printf.o assert.o int_to_string.o lpc2106.o chrono.o
rtc.o keyboard.o

ifdef ROM
%.o: %.c
     arm-eabi-gcc -O3 -g -c -I$(HOME)/mylib -I$(HOME)/clib/inc \
          --save-temps -o $*.o $<

%.o: %.s
     arm-eabi-as --gstabs -a=$*.lst -o $*.o $<

main.bin: main.elf
```

```
    arm-eabi-objcopy -Obinary main.elf main.bin

main.elf: cstart_rom.o $(OBJECTS)
    arm-eabi-ld cstart_rom.o $(OBJECTS) -o main.elf \
        -T$(HOME)/mylib/ldscript_rom -L$(HOME)/clib/lib -lc $(LIBGCC)
else

%.o: %.c
    arm-eabi-gcc -g -c -I$(HOME)/mylib -I$(HOME)/clib/inc \
        --save-temps -o $*.o $<

%.o: %.s
    arm-eabi-as --gstabs -a=$*.lst -o $*.o $<

main.elf: cstart_ram.o $(OBJECTS)
    arm-eabi-ld cstart_ram.o $(OBJECTS) -o main.elf \
        -T$(HOME)/mylib/ldscript_ram -L$(HOME)/clib/lib -lc $(LIBGCC)
endif

.PHONY: clean
clean:
    -rm -f  *.s *.elf *.exe.* *.bin *.o *.lst *.bak *.map *.ii *.i *.d *~
```

makefile