Combining Executable Files for ROMing

Proposal

Revision History

Version	Primary Author(s)	Description of Version	Date Completed
v0.1	Anbu Gopalrajan Alan Davis	Initial Version	1/9/2008
V1.0	*Anbu Gopalrajan*	*Added more detailed examples*	*2/11/2008*
V1.1	*Cody Addison*	*Minor modifications and corrections*	*9/02/2008*
V1.2	*Cody Addison*	*Changed to reflect implemented behavior*	*10/24/2008*

Copyright © Texas Instruments 2008

Table of Contents

Copyright © Texas Instruments 2008

1 Overview

TI customers use the TI compiler toolset to build application(s) into an executable file that can be loaded and executed on the TI devices. The result of this application build process is usually a single executable. If an RTOS is used, it is statically linked in with the applications to create a single executable. This executable is loaded on the device for execution.

There are many ways to load the executable on the device. One way is to ROM mask the executable's load image on the device. Such ROM image can be executed in place or copied to RAM and executed. Currently the customers provide a single executable file to TI for ROM masking. The load image is represented in the executable file by initialized sections and their load addresses. Data from the initialized sections are masked in the device ROM at their specific load addresses.

Creating a single executable for ROM masking is not the only application development model anymore. Now executables are built and tested to run on multiple sockets. Customers build applications, boot loader and configuration data as separate executables and test them. Then they choose a set of executables as complete application to be ROM masked. This means multiple executable files are to be delivered for ROM masking. Though the ROM masking process can handle multiple executables for a single socket, the customers want to retain control over how these executables are combined for ROM masking. A mistake in ROM masking is very expensive and the customers want to avoid any potential problems. Delivering a single file for ROM masking has worked well and customers want to combine these executables at their end and deliver a single file for ROM masking.

The TI linker links in object files to create an executable. However, it does not support linking in multiple executables as required by the above use case. The basic functionality of the linker is to perform symbolic linkage, place the input sections in defined memory and perform relocation. When linking in the executables for the above use case, the requirement is to **not** perform symbolic linkage, placement and relocation. The sections in the input executables should be copied 'as is' to the output. There is no clean way to support this in the linker. Also, the linker copies global symbols to the output file and there is no clear way to handle multiple global symbol definitions. Hence the linker is not the best tool to handle this use case.

The executable COFF or ELF file has much more information than what is required for ROM masking. For example, the symbol table, the relocation entries and the uninitialized sections are not used during ROM masking. Only the initialized sections and their load addresses are required. If we can copy all the initialized sections from the executables into a single file then such a file can be delivered for ROM masking. That is, tool support is needed to combine the load images from multiple executables into a single file. Since currently TI COFF or ELF files are delivered for ROM masking, the combined load image file should use TI COFF or ELF object format.

Converting an executable file into a load image object file has other applications. For example, such an object file can be used to include an executable as a boot image in a subsequent link step.

This document outlines a proposal to combine multiple executables to create a single load image object file.

2 Requirements

The basic requirement is to accept one or more executables and create a single load image file. The load image file should use TI COFF or ELF object format.

Copyright © Texas Instruments 2008

Currently the TI assembler can generate relocatable object files in TI COFF or ELF object format. The linker can accept these files to create an executable file in the same object format. The load image object file differs from these files in the following ways:

- It contains the file header, the section headers and one or more initialized data sections.
- It does not contain global symbols
- It does not contain any relocation entries. That is, relocation cannot be performed on these files.
- It does not contain any uninitialized sections or executable sections. It just represents the load image of an executable using the initialized data section(s).
- It does not have an entry point and is not marked an executable
- It does not have any debug information.
- It can be linked in a subsequent link step. However, it cannot be relocated. Only the load address can change in the subsequent link step.

The following lists the user requirements:

1. Accept one or more executable files and create a single TI COFF or ELF load image object file.
2. The initialized sections from the executable files are copied to the load image file.
3. Symbols, relocation entries, entry points and uninitialized sections from the executables are discarded.
4. The initialized sections can be combined into a single load image section if they are contiguous. Contiguous sections are determined using the load address.
5. Alternatively, the user can specify a memory range to be represented as a single load image section. In this case, all the initialized sections in this memory range are copied and any holes are filled using a default value or a user specified value.
6. The load address is preserved in the load image object file. Run address is discarded.
7. The user should be able to name the sections in the load image object file.
8. The output object file is not marked an executable. It should be linkable in a subsequent link step.
9. Generate diagnostics for overlapping memory ranges.

3 Proposal

The hex converter accepts an executable file and generates hex data from the initialized sections. It supports specifying a memory range and has the basic framework to create the load image file. Our proposal is to enhance the hex converter to generate the load image object file in TI COFF or ELF format from one or more executable files.

Copyright © Texas Instruments 2008

Please refer to the Assembly Language Tools User's Guide (SPNU118D) for more details on hex converter.

The following specific changes are proposed.

3.1 Load Image Object File Generation

The hex converter supports many output formats: TI-Tagged, ASCII-hex, Intel, Motorola-S, Tektronix, etc. The proposal is to add support to generate a load image object file. The load image object file's format is determined by the input files. The load image format can be invoked with the following option

--load_image ➔ Create a load image object file with format the same as the input files

3.2 Accept Multiple Input Files

The hex converter accepts a single object file now. The proposal is to make the hex converter accept one or more input object files. The following restrictions apply:

- ELF and TI COFF files cannot be mixed as input files.
- Accepting multiple input files will be available for all currently supported formats.
- SECTIONS directive will be extended to support multiple files

The SECTIONS directive has the following format:

```
SECTIONS
{
      sname: paddr=<value>
}
```

The directive will be extended to support multiple files by allowing the file name to be specified along with the section name. This is may be useful in the case of generating load image objects, but it is needed if multiple file support is added for all output formats. The format is as follows:

```
SECTIONS
{
      fname(sname): paddr=<value>
}
```

3.3 Load Image Section Formation

The load image sections are formed by collecting the initialized sections from the input executables. There are two ways the load image sections are formed as described below.

[ASIDE: TI COFF executables only contain sections. ELF executables on the other hand contain segments and need not have sections. However, the TI ELF executables contain both segments and sections. Hence, in this proposal, we consider input sections from ELF executables for copying into load image sections. When an ELF executable only contains segments, the segments are used. However, the SECTION directive of the HEX converter will be ignored.]

3.3.1 Using Address Ranges

The hex converter supports the ROMS directive to let the user specify an address range. The ROMS directive has the following format:

Copyright © Texas Instruments 2008

```
ROMS
{
      romname : [origin=value,] [length=value,] [romwidth=value,]
                [ memwidth=value,] [fill=value] [files={filename1, filename2, ...}]

...

}
```

When using –load_image, the –image option must be specified. If it is not, an error will be generated. The keywords *origin* and *length* define an address range in memory. This memory range is given a name (romname above). When the user specifies the ROMS directive to build a load image object, a single load image section is created for this memory range. Any holes in this range are filled using the default value zero. Alternatively, the user can use the keyword *fill* to specify a fill value.

When the ROMS directive is used, any input section with load address outside the defined range is discarded with a warning.

3.3.2 Default Load Image Section Formation

When no ROMS directive is used, the load image sections are formed by combining contiguous initialized sections in the input executables. The hex converter will consider input sections with holes smaller than target word size to be contiguous. For example, assume a TMS470 target which has 4-byte target word. Also assume an input section ends at address 0x3FFD and the next input section starts at 0x4000. Though there is a hole of 2 bytes, these input sections are considered contiguous.

3.4 Load Image Section Name

The user wants to control the name given to the load image section. As we discussed above, there are two ways to form the load image sections. The load image section name in each case can be controlled differently.

3.4.1 When using ROMS directive

As mentioned above, a ROMS directive specifies memory range which results in a single load image section. Also each memory range has to have a name (romname above). This memory range name can be used to name the load image section. One issue is that currently the hex converter limits this name to 32 characters. Our proposal is to extend the hex converter to support arbitrary length memory range name.

```
ROMS
{
      App1 : [origin=value,] [length=value,]

}
```

3.4.2 When using default section formation

In the absence of ROMS directive, the hex converter uses default load image section formation by collecting contiguous initialized sections. In this model, the user can specify a section prefix using a command line option. Our proposal is to add --section_name_prefix=<prefix> option. When this option is specified, the hex converter names the load image sections by suffixing a running number to <prefix>.

Copyright © Texas Instruments 2008

For example, '--section_name_prefix=image' option names the load image sections image_1, image_2, etc.

This option is ignored if not generating TI COFF/ELF load image output.

3.5 COFF/ELF Load Image Section Characteristics

1. It is always an initialized data section
2. In the absence of ROMS directive, the load/run address of the load image section is the load address of the first input section in the load image section. If the ROMS directive is used to specify a memory range, then the load/run address of the load image section is the origin of the memory range.
3. The name of the load image section is:
    a. The ROMS memory range name, if the ROMS directive is specified
    b. Else,
        i. <prefix> suffixed with a running number if --section_name_prefix=<prefix> is specified
        ii. else the default section name prefix is used (proposed default is "image")

3.6 COFF/ELF Load Image File Characteristics

1. A single object file will be generated
2. The object file is not marked an executable
3. Symbols and relocation entries from the input files are not copied
4. The output load image file is named as follows:
    a. If -o option is specified, it is used.
    b. Otherwise, default file name is used. Default file name is ti_load_image.obj
    c. The *files* keyword in the ROMS directive will be ignored. If present a warning will be generated.

3.7 Memory width and ROM width

The hex converter allows the user to specify a memory width and a ROM width. Please refer to the Assembly Language Tools User's Guide (SPNU118D) for more details. A ROM width smaller than a memory width results in the hex converter splitting memory width bits from input into ROM width bits in multiple output files. The TI COFF/ELF object files contain raw data and there is no way to specify the size of each word. Hence, when generating TI load image object file, both the ROM width and memory are ignored. If the memory width or ROM width is specified, a warning will be issued informing the user that the values are being ignored.

Copyright © Texas Instruments 2008

4 Example

This section contains three examples. The first example shows a simple use case where three executables are combined. The second example shows how the user can define memory ranges to control the output load image section formation. The third example shows building a boot loader with embedded boot image.

4.1 Example 1: A simple use case

Below, we show how three executables are combined using the default load section formation. The three executables in this example are an application (app.out), a boot loader (boot.out) and a configuration data out file (config.out). The contents of these executables are given below. Note that each executable contains contiguous set of initialized sections which form a single contiguous initialized area in load memory. For example, the app.out has three initialized sections, namely .text, .cinit and .const, and they are contiguous in memory from 0x20 through 0x4955.

```
> ofd470 app.out

OBJECT FILE:  app.out
...
Section Information
    id name                      load addr  run addr     size align alloc
    -- ----                      ---------  --------     ---- ----- -----
     3 .bss                     0x0005a000 0x0005a000  0x478     4   Y
     4 .sysmem                  0x00058000 0x00058000 0x2000     4   Y
     5 .stack                   0x00050000 0x00050000 0x8000     4   Y
     6 .text                    0x00000020 0x00000020 0x4700     4   Y
     7 .cinit                   0x00004720 0x00004720  0x134     4   Y
     8 .const                   0x00004854 0x00004854  0x101     4   Y
...

> ofd470 boot.out

OBJECT FILE:  boot.out
...
 Section Information
    id name                      load addr  run addr     size align alloc
    -- ----                      ---------  --------     ---- ----- -----
     3 .bss                     0x00050500 0x00050500  0x256     4   Y
     4 .sysmem                  0x0005A000 0x0005A000 0x2000     4   Y
     5 .stack                   0x00060000 0x00060000 0x8000     4   Y
     6 .text                    0x00005000 0x00005000  0x954     4   Y
     7 .cinit                   0x00005954 0x00005954   0x48     4   Y
     8 .const                   0x0000599C 0x0000599C   0x80     4   Y
...

> ofd470 config.out

OBJECT FILE:  config.out
...
Section Information
    id name                      load addr  run addr     size align alloc
    -- ----                      ---------  --------     ---- ----- -----
     3 .config                  0x00007000 0x00007000 0x37a0     4   Y
...
```


Copyright © Texas Instruments 2008

The customers want to deliver the above executables for ROM masking and they want to deliver a single object file to make sure the components don't interfere.

- Combine the executables to create the output file 'ROM_simple.obj'.
- Use the default load image section formation.
- Name the load image data sections data_1, data_2, etc.

```
> hex470 app.out boot.out config.out --section_name_prefix="data"
--load_image -o ROM_simple.obj

> ofd470 ROM_simple.obj

OBJECT FILE:  ROM_simple.obj
 ...
 Section Information
    id name                      load addr  run addr    size align alloc
    -- ----                      ---------  --------    ---- ----- -----
     1 data_1                    0x00000020 0x00000020  0x4938  4    Y
     2 data_2                    0x00005000 0x00005000  0x0a1c  4    Y
     3 data_3                    0x00007000 0x00007000  0x37a0  4    Y
...
```

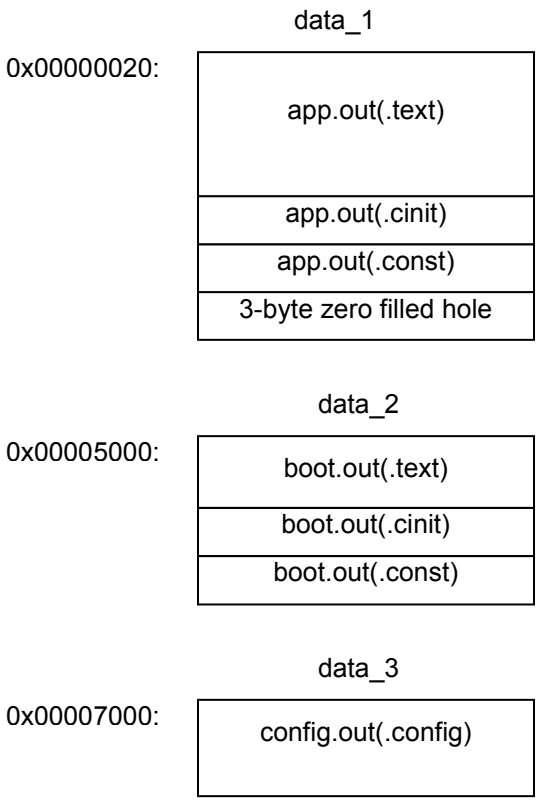

Fig 4.1 ROM_simple.obj Layout

Copyright © Texas Instruments 2008

4.2 Example 2: User control of load image section formation

Again, let us consider three executables, namely an application (app.out), a boot loader (boot.out) and configuration data (config.out). The contents of these executables are shown below. Note that the initialized sections in apps.out are not contiguous. But the user wants to get a single load image data section for each input executable. To accomplish this the user defines three memory ranges, one for each input execuable.

```
> ofd470 app.out

OBJECT FILE:  app.out
...
Section Information
    id name                      load addr  run addr     size align alloc
    -- ----                      ---------  --------     ---- ----- -----
     3 .bss                      0x0005a000 0x0005a000  0x478     4   Y
     4 .sysmem                   0x00058000 0x00058000 0x2000     4   Y
     5 .stack                    0x00050000 0x00050000 0x8000     4   Y
     6 .text                     0x00000020 0x00000020 0x4700     4   Y
     7 .cinit                    0x00004800 0x00004800  0x134     4   Y
     8 .const                    0x00004934 0x00004934  0x100     4   Y
...

> ofd470 boot.out

OBJECT FILE:  boot.out
...
 Section Information
    id name                      load addr  run addr     size align alloc
    -- ----                      ---------  --------     ---- ----- -----
     3 .bss                      0x00050500 0x00050500  0x256     4   Y
     4 .sysmem                   0x0005A000 0x0005A000 0x2000     4   Y
     5 .stack                    0x00060000 0x00060000 0x8000     4   Y
     6 .text                     0x00005000 0x00005000  0x954     4   Y
     7 .cinit                    0x00005954 0x00005954   0x48     4   Y
     8 .const                    0x0000599C 0x0000599C   0x80     4   Y
...

> ofd470 config.out

OBJECT FILE:  config.out
...
Section Information
    id name                      load addr  run addr     size align alloc
    -- ----                      ---------  --------     ---- ----- -----
     3 .config                   0x00007000 0x00007000 0x37a0     4   Y
...
```

Given the above executables,

- Combine them to create the output file ‘ROM.obj’.
- Each executable has its own memory range. Specify a memory range for each executable.
- The load image data section corresponding to the application, boot loader and configuration data executables are named apps, bootloader and config respectively.

Copyright © Texas Instruments 2008

```
> cat rom.cmd

app.out         /* Apps input file          */
boot.out        /* Boot loader input file   */
config.out      /* Config input file        */

--load_image  /* TI COFF load image output format */
-o ROM.obj            /* combined load image object  */

ROMS
{
      apps:       origin=0x00000020 len=0x4FE0
      bootloader: origin=0x00005000 len=0x1000
      config:     origin=0x00007000 len=0x4000 fill=0xABCD
}

> hex470 rom.cmd

> ofd470 ROM.obj

OBJECT FILE:  ROM.obj
 ...
 Section Information
    id name                      load addr  run addr    size align alloc
    -- ----                      ---------  --------    ---- ----- -----
     1 apps                      0x00000020 0x00000020  0x4FE0  4    Y
     2 bootloader                0x00005000 0x00005000  0x1000  4    Y
     3 config                    0x00007000 0x00007000  0x4000  4    Y
...
```


Copyright © Texas Instruments 2008

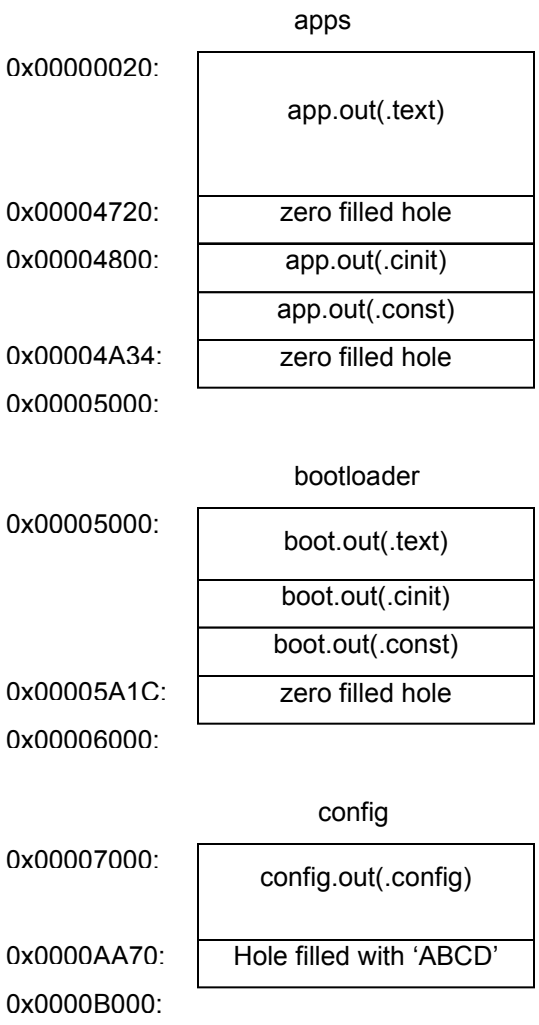

Fig 4.2 ROM.obj Layout

4.3 Example 3: Build boot loader with embedded boot image

This example shows how the user can build a boot loader with embedded boot image as a single executable. The executable app.out is converted into a load image and included in a subsequent link step to produce a single executable containing the boot loader and the application boot image. Note that the initialized sections in the executable app.out are contiguous.

```
> ofd470 app.out

OBJECT FILE:  app.out
...
Section Information
    id name                     load addr  run addr     size align alloc
    -- ----                     ---------  --------     ---- ----- -----
     3 .bss                    0x0005a000 0x0005a000  0x478     4   Y
     4 .sysmem                 0x00058000 0x00058000 0x2000     4   Y
     5 .stack                  0x00050000 0x00050000 0x8000     4   Y
```


Copyright © Texas Instruments 2008

```
     6 .text                      0x00000020 0x00000020 0x4700     4   Y
     7 .cinit                     0x00004720 0x00004720  0x134     4   Y
     8 .const                     0x00004854 0x00004854  0x101     4   Y
...
```

Step 1:

Create the boot image of the application.

```
> hex470 app.out --section_name_prefix="app_image" --load_image -o app_image.obj

> ofd470 app_image.obj

OBJECT FILE:  app_image.obj
 ...
 Section Information
    id name                      load addr  run addr    size align alloc
    -- ----                      ---------  --------    ---- ----- -----
     1 app_image_1               0x00000020 0x00000020  0x4938  4   Y
...
```

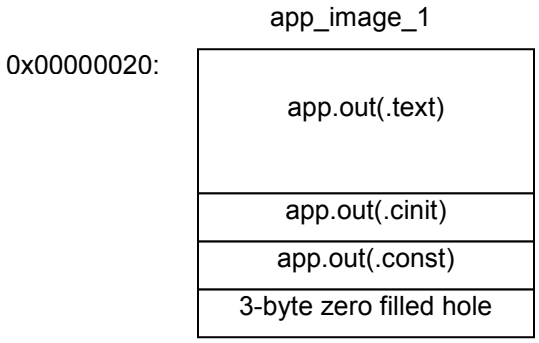

Fig 4.3 app_image.obj layout

Step 2:

Include the object app_image.obj in the boot loader link step. Note that the load image section app_image_1 can be loaded at any address but should be copied to the run address 0x00000020 before executing it.

```
> cat lnk.cmd

-c                                        /* LINK USING C CONVENTIONS      */
--stack_size  0x8000                      /* SOFTWARE STACK SIZE           */
-heap   0x2000                            /* HEAP AREA SIZE                */

/* SPECIFY THE SYSTEM MEMORY MAP */
MEMORY
{
    APP_MEM   : org = 0x00000020   len = 0x000040E0  /* APP RUN MEMORY (RAM) */
    P_MEM     : org = 0x00005000   len = 0x00040000  /* PROGRAM MEMORY (ROM) */
```


Copyright © Texas Instruments 2008

```
    D_MEM    : org = 0x00045000   len = 0x00050000  /* DATA MEMORY     (RAM) */
}

/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */

SECTIONS
{
    .intvecs : {} > 0x0                /* INTERRUPT VECTORS                  */
    .bss     : {} > D_MEM              /* GLOBAL & STATIC VARS               */
    .sysmem  : {} > D_MEM              /* DYNAMIC MEMORY ALLOCATION AREA     */
    .stack   : {} > D_MEM              /* SOFTWARE SYSTEM STACK              */

    .text    : {} > P_MEM              /* CODE                               */
    .cinit   : {} > P_MEM              /* INITIALIZATION TABLES              */
    .const   : {} > P_MEM              /* CONSTANT DATA                      */

    .pinit   : {} > P_MEM              /* C++ CONSTRUCTOR TABLES             */
    .xref    : > P_MEM
    .app_image: {*(app_image*)} > LOAD=P_MEM, RUN=APP_MEM,
LOAD_START(_app_load_start), RUN_START(_app_run_start), SIZE(_app_size)
}


>lnk470 boot_*.obj app_image.obj lnk.cmd -o boot.out
```

The output file boot.out contains the boot loader and the application boot image. The boot loader uses the symbols _app_load_start, _app_run_start and _app_size to copy the boot image from load address to run address.

Copyright © Texas Instruments 2008