ARM ELF File Format

ARM DUI 00101-A

Release Information

The following changes have been made to this book.

Change History

Date	Issue	Change
Nov 1998	A	Created.

Overview of ELF File Format

This document describes the ARM implementation of the ELF file format used in the ARM Software Development Toolkit version 2.50. It is assumed that the reader is familiar with ELF version 1. This section only describes options taken by ARM in its executable file format. Unless otherwise stated, Executable ARM ELF files are as defined in the TIS Portable Formats Specification, Version 1.1.

Object file format

ELF describes three types of Object File:

- relocatable file
- executable file
- shared object file.

In general, an ELF Object File has the following organization:

Linking View		**Execution View**
`ELF Header`		`ELF Header`
`Program Header Table (Optional)`		`Program Header Table`
`Section 1`		`Segment 1`
`....`		
`Section n`		`Segment 2`
`....`		
`....`		`....`
`Section Header Table`		`Section Header Table (Optional)`

Section header table

The view of an Object File as a series of named Sections is used by a linker or debugger. Several Sections are denoted *special* and have reserved names. For example:

- .symtab
- .strtab

The Section Header Table gives access to such sections.

Program header table

The view of an Object File as a series of Segments is typically used by a loader in order to create an executable process image for a particular runtime environment.

The Program Header Table gives access to such Segments.

Executable ARM ELF File Layout

The ARM linker is used to produce an Executable ARM ELF file. For simple cases, it lays out the file as shown in *Generic ELF File Layout* on page 4.

Extra information is encoded in the file for scatter-loaded and overlayed executables and these special cases are described in *Scatter-loaded Executables* on page 10.

Only Segments will form part of the final executable image. Sections are included in the Executable to provide further information about the executable image.

Generic ELF File Layout

A simple Executable ARM ELF file has the conceptual layout shown in the diagram on the right.

Note that the actual ordering of the file may be different from that shown, since only an ELF header has a fixed position in the file.

All other parts of the file have a position defined by:

- the ELF header
- the Program Header Table
- the Section Header Table.

`ELF Header`
`Program Header Table`
`Text segment`
`Data segment`
`BSS segment`
`".symtab" section`
`".strtab" section`
".shstrtab" section
Debug sections
`Section Header Table`

ARM-specific ELF Header Values

This section describes the values in the ELF header which need to be defined for the ARM target environment. All other values are as specified in the *Tool Interface Standard Portable Formats Specification*:

`e_machine`	is set to `EM_ARM` (defined as 40)
`e_ident[EI_CLASS]`	is set to `ELFCLASS32`
`e_ident[EI_DATA]`	is set to: `ELFDATA2LSB` for little-endian targets `ELFDATA2MSB` for big-endian targets

Note

The endianness of the target is determined by the endianness of the Object Files submitted to the ARM linker. The linker will produce an error message if presented with object files of mixed endianness.

Segments

There are three types of Segment:

- Text
- Data
- BSS

Entries for these appear in the Program Header Table.

In a simple Executable ARM ELF file, there is just one of each type of Segment. More complex cases are described in *Scatter-loaded Executables* on page 10.

Attributes of these Segments are described below:

Text Segment

Contains the code for the executable.

```
p_type        - set to PT_LOAD
p_vaddr       - load address of the segment
p_paddr       - 0
p_filesz      - size of text segment
p_memsz       - same as p_filesz
p_flags       - PF_X + PF_R
p_align       - 4
```

Data Segment

Contains initialized read-write data for the executable.

```
p_type        - set to PT_LOAD
p_vaddr       - load address of data segment
p_paddr       - 0
p_filesz      - size of data segment
p_memsz       - same as p_filesz
p_flags       - PF_R + PF_W
p_align       - 4
```

BSS Segment

Contains uninitialized data, which should be zeroed either when an image is created, or at program startup by the runtime environment. Note that a BSS Segment is distinguished by having a p_filesz of 0 to indicate that it occupies no space in the executable file.

```
p_type        - set to PT_LOAD
p_vaddr       - load address of BSS data segment
p_paddr       - 0
p_filesz      - 0  (note: occupies no file space)
p_memsz       - size of BSS segment
p_flags       - PF_R + PF_W
p_align       - 4
```

Sections

Under the ELF specification, an executable object file can include a Section Header Table which describes Sections in the file. In Executable ARM ELF, all Executables have at least two Sections, unless the linker has been invoked with `-nodebug`:

- the Symbol Table Section
- the String Table Section.

Further Sections may appear in the file, and these are described later in *Scatter-loaded Executables* on page 10.

When an Executable contains source-level debugging information, it also includes several Debugging Sections, as described below.

If required, an Executable can be stripped of its Sections, leaving just the Text, Data and BSS Segments. The Section Header Table is also removed.

Symbol Table Section

The Symbol Table Section has the following attributes:

```
sh_name:   ".symtab"
sh_type:   SHT_SYMTAB
sh_addr:   0  (to indicate it is not part of the image)
```

Note

In Executable ARM ELF we do not set the `SHF_ALLOC` bit in the `sh_flags` field, thus indicating that there is no space allocated for the symbol table in the image which will be created from this Executable.

This symbol table can be used for low-level debugging symbol information.

String Table Section

The String Table Section holds all strings referenced by other Sections in the Executable. In particular it will hold the textual names of entries in the Symbol Table Section. It has the following attributes:

```
sh_name:   ".strtab"
sh_type:   SHT_STRTAB
sh_addr:   0 (to indicate it is not part of the image)
```

Section Name String Table

The Section Name String Table holds the textual names of all sections. It has the following attributes:

```
sh_name:   ".shstrtab"
sh_type:   SHT_STRTAB
sh_addr:   0 (to indicate it is not part of the image)
```

Debugging Sections

ARM Executable ELF supports three types of debugging information held in debugging Sections. A consumer of an ELF executable can distinguish between these three types of debugging information by examining the Section Table for the executable:

- ASD debugging tables

  These provide backwards compatibility with ARM's Symbolic Debugger. ASD debugging information is stored in a single Section in the executable named `.asd`.

- DWARF version 1.0

  When DWARF 1.0 debugging information is included by the linker in the ELF executable, the file contains the following ELF Sections, each of which has a Section Header Table entry:

Section name	**Contents**
`.debug`	debugging entries
`.line`	`fileinfo` entries
`.debug_pubnames`	table for accelerated access to debug items
`.debug_aranges`	address ranges for compilation units

- DWARF version 2.0

  When DWARF 2.0 debugging information is included by the linker in the ELF executable, the file contains the following ELF sections, each of which has a Section Header Table entry:

Section name	**Contents**
`.debug_info`	debugging entries
`.debug_line`	`fileinfo` statement program
`.debug_pubnames`	table for accelerated access to debug items
`.debug_aranges`	address ranges for compilation units
`.debug_macinfo`	macro information (#define / #undef)
`.debug_frame`	call frame information
`.debug_abbrev`	abbreviation table
`.debug_str`	debug string table

Each of the `.debug_*` sections will have type SHT_PROGBITS. These are only included when source-level debugging information is available.

Each entry will have a `sh_addr` member of 0 indicating that the file contains the debugging information, but that this information will not be included in an image created from the executable.

Note

This means debugging information (albeit maybe only low-level debug symbols) can be kept for a program image residing in ROM without that information appearing in the ROM itself.

Scatter-loaded Executables

When scatter loading is used, the ARM linker generates Section Header Table entries for a load region, where the Section name is taken from the load region name as defined in the scatter description.

If the load region contains a mixture of code, data and uninitialized data, there will be more than one Segment generated for that load region:

- each Segment generated will have its own Section Header Table entry, so more than one Section may have the same name
- each Section Header Table entry will have its sh_offset field set to the same value as the p_offset field of the Program Header Table entry for its corresponding Segment.

Each Section Header Table entry for a load region will have the following attributes:

`sh_name:`	name of load region (as given in scatter description)
`sh_type:`	SHT_PROGBITS or SHT_NOBITS (for zero init areas)
`sh_addr:`	same as p_vaddr of corresponding Segment
`sh_offset:`	same as p_offset of corresponding Segment
`sh_flags:`	bit SHF_LOADREGION set

The `p_vaddr` field of each Segment of a scatter-loaded Executable is the load address of the Segment, which need not necessarily be its execution address. Startup code can move (part of) a Segment to its execution address using the symbols:

```
Load$$reg$$Base
Image$$reg$$Base
Image$$reg$$Length
```

as described in the *Software Development Toolkit User Guide.*

The Section Header Table entries can be used to generate a separate plain binary file for each load region, using the Section name as the name of the generated file. A tool which does this would need to merge any Sections of the same name, sorting them by address (`sh_addr`). Alternatively a single image can be made simply by using the Segment information held in the Program Header Table.

Note

Debugging information (if included) for the Executable will refer to memory locations in execution regions rather than load regions.