Chapter 13
ARM Image Format

This chapter describes the ARM Image Format (AIF). It contains the following sections:

13.1 Overview of the ARM Image Format

ARM Image Format (AIF) is a simple format for ARM executable images, consisting of:

- a 128-byte header
- the image code
- the image initialized static data.

An AIF image is capable of self-relocation if it is created with the appropriate linker options. The image can be loaded anywhere and it will execute where it is loaded. After an AIF image has been relocated, it can create its own zero-initialized area. Finally, the image is entered at the unique entry point.

13.2 AIF variants

There are three variants of AIF:

Executable AIF

Executable AIF can be loaded at its load address and entered at the same point (at the first word of the AIF header). It prepares itself for execution by relocating itself if required and setting to zero its own zero-initialized data.

The header is part of the image itself. Code in the header ensures that the image is properly prepared for execution before being entered at its entry address.

The fourth word of an executable AIF header is:

`BL` *`entrypoint`*

The most significant byte of this word (in the target byte order) is 0xeb.

The base address of an executable AIF image is the address at which its header should be loaded. Its code starts at *base* + 0x80.

Non-executable AIF

Non-executable AIF must be processed by an image loader that loads the image at its load address and prepares it for execution as detailed in the AIF header. The header is then discarded. The header is not part of the image, it only describes the image.

The fourth word of a non-executable AIF image is the offset of its entry point from its base address. The most significant nibble of this word (in the target byte order) is 0x0.

The base address of a non-executable AIF image is the address at which its code should be loaded.

Extended AIF

Extended AIF is a special type of non-executable AIF. It contains a scatter-loaded image. It has an AIF header that points to a chain of load region descriptors within the file. The image loader should place each region at the location in memory specified by the load region descriptor.

13.3 The layout of AIF

This section describes the layout of AIF images.

13.3.1 AIF image layout

An AIF image has the following layout:

- Header
- Read-only area
- Read-write area
- Debugging data (optional)
- Self-relocation code (position-independent)
- Relocation list. This is a list of byte offsets from the beginning of the AIF header, of words to be relocated, followed by a word containing `-1`. The relocation of non-word values is not supported.

Note

An AIF image is restartable if, and only if, the program it contains is restartable (an AIF image is *not* reentrant). Following self-relocation, the second word of the header must be reset to `NOP`. This causes no additional problems with the read-only nature of the code section.

On systems with memory protection, the self-relocation code must be bracketed by system calls to change the access status of the read-only section (first to writable, then back to read-only).

13.3.2 Debugging data

After the execution of the self-relocation code, or if the image is not self-relocating, the image has the following layout:

- Header
- Read-only area
- Read-write area
- Debugging data (optional).

AIF images support being debugged by an ARM debugger. Low-level and source-level support are orthogonal. An AIF image can have both, either, or neither kind of debugging support.

References from debugging tables to code and data are in the form of relocatable addresses. After loading an image at its load address these values are effectively absolute.

References between debugger table entries are in the form of offsets from the beginning of the debugging data area. Following relocation of a whole image, the debugging data area itself is position-independent and may be copied or moved by the debugger.

13.3.3 AIF header

Table 13-1 shows the layout of the AIF header.

Table 13-1 AIF header layout

00:	NOP[a]	
04:	BL SelfRelocCode	NOP if the image is not self-relocating
08:	BL ZeroInit	NOP if the image has none.
0C:	BL ImageEntryPoint or EntryPoint Offset	BL to make the header addressable via r14 ...but the application will not return... Non-executable AIF uses an offset, not BL. BL is used to make the header addressable via r14 in a position-independent manner, and to ensure that the header will be position-independent.
10:	Program Exit Instruction	… last attempt in case of return. The Program Exit Instruction is usually a SWI causing program termination. On systems that do not implement a SWI for this purpose, a branch-to-self is recommended. Applications are expected to exit directly and *not* to return to the AIF header, so this instruction should never be executed. The ARM linker sets this field to SWI 0x11 by default, but it may be set to any desired value by providing a template for the AIF header in an area called AIF_HDR in the *first* object file in the input list to armlink.
14:	Image ReadOnly size	Image ReadOnly Size includes the size of the AIF header only if the AIF type is executable (that is, if the header itself is part of the image).
18:	Image ReadWrite size	Exact size (a multiple of 4 bytes).
1C:	Image Debug size	Exact size (a multiple of 4 bytes). Includes high-level and low-level debug size. Bits 0-3 hold the type. Bits 4-31 hold the low level debug size.
20:	Image zero-init size	Exact size (a multiple of 4 bytes).

Table 13-1 AIF header layout (Continued)

24:	Image debug type	Valid values for Image debug type are: **0** No debugging data present. **1** Low-level debugging data present. **2** Source level debugging data present. **3** 1 and 2 are present together. All other values of image debug type are reserved.
28:	Image base	Address where the image (code) was linked.
2C:	Work space	Obsolete.
30:	Address mode: 26/32 + 3 flag bytes	The word at offset 0x30 is 0, or contains in its least significant byte (using the byte order appropriate to the target): **26** Indicates that the image was linked for a 26-bit ARM mode, and may not execute correctly in a 32-bit mode. This is obsolete. **32** Indicates that the image was linked for a 32-bit ARM mode, and may not execute correctly in a 26-bit mode. A value of 0 indicates an old-style 26-bit AIF header. If the Address mode word has bit 8 set, the image was linked with separate code and data bases (usually the data is placed immediately after the code). The word at offset 0x34 contains the base address of the image's data.
34:	Data base	Address where the image data was linked.
38:	Two reserved words (initially 0)	In Extended AIF images, the word at 0x38 is non-zero. It contains the byte offset within the file of the header for the first non-root load region. This header has a size of 44 bytes, and the following format: **word 0** file offset of header of next region (0 is none) **word 1** load address **word 2** size in bytes (a multiple of 4) **char[32]** the region name padded out with zeros. The initializing data for the region follows the header.
40:	`NOP`	
44:	Zero-init code 15 words as below	Header is 32 words long.

a. In all cases, `NOP` is encoded as `MOV r0,r0`

Chapter 14
ARM Object Library Format

This chapter describes the ARM Object Library Format (ALF). It contains the following sections:

14.1 Overview of ARM Object Library Format

This section defines a file format called *ARM Object Library Format* (ALF), that is used by the ARM linker and the ARM object librarian.

A library file contains a number of separate but related pieces of data. In order to simplify access to these data, and to provide for a degree of extensibility, the library file format is itself layered on another format called *Chunk File Format*. This provides a simple and efficient means of accessing and updating distinct chunks of data within a single file. Refer to *Chunk file format* on page 15-4 for a description of the Chunk File Format.

The Library format defines four chunk classes:

- Directory
- Time stamp
- Version
- Data.

There may be many *Data* chunks in a library.

The Object Library Format defines two additional chunks:

- Symbol table
- Symbol table time stamp.

14.2 Endianness and alignment

For data in a file, *address* means *offset from the start of the file*.

There is no guarantee that the endianness of an ALF file will be the same as the endianness of the system used to process it (the endianness of the file is always the same as the endianness of the target ARM system).

The two sorts of ALF cannot meaningfully be mixed (the target system cannot have mixed endianness, it must have one or the other). The ARM linker accepts inputs of either sex and produces an output of the same sex, but rejects inputs of mixed endianness.

14.2.1 Alignment

Strings and bytes may be aligned on any byte boundary.

ALF fields defined in this document do not use halfwords, and align words on 4-byte boundaries.

Within the contents of an ALF file (within the data contained in `OBJ_AREA` chunks, see below), the alignment of words and halfwords is defined by the use to which ALF is being put. For all current ARM-based systems, alignment is strict.

14.3 Library file format

For library files, the first part of each chunk name is `LIB_`. For object libraries, the names of the additional two chunks begin with `OFL_`.

Each piece of a library file is stored in a separate, identifiable chunk. Table 14-1 shows the chunk names.

Table 14-1 Library File Chunks

Chunk	Chunk name
Directory	LIB_DIRY
Time stamp	LIB_TIME
Version	LIB_VRSN
Data	LIB_DATA
Symbol table	OFL_SYMT object code
Time stamp	OFL_TIME object code

There may be many `LIB_DATA` chunks in a library, one for each library member. In all chunks, word values are stored with the same byte order as the target system. Strings are stored in ascending address order, which is independent of target byte order.

14.3.1 Earlier versions of ARM object library format

These notes ensure maximum robustness with respect to earlier, now obsolete, versions of the ARM object library format:

- Applications which create libraries or library members should ensure that the `LIB_DIRY` entries they create contain valid time stamps.
- Applications which read `LIB_DIRY` entries should not rely on any data beyond the end of the name string being present, unless the difference between the DataLength field and the name-string length allows for it. Even then, the contents of a time stamp should be treated cautiously.
- Applications which write `LIB_DIRY` or `OFL_SYMT` entries should ensure that padding is done with NULL (0) bytes. Applications that read `LIB_DIRY` or `OFL_SYMT` entries should make no assumptions about the values of padding bytes beyond the first, string-terminating NULL byte.

14.3.2 LIB_DIRY

The LIB_DIRY chunk contains a directory of the modules in the library, each of which is stored in a LIB_DATA chunk. The directory size is fixed when the library is created. The directory consists of a sequence of variable length entries, each an integral number of words long. The number of directory entries is determined by the size of the LIB_DIRY chunk. Table 14-2 shows the layout.

Table 14-2 The LIB_DIRY chunk

ChunkIndex	
EntryLength	The size of this LIB_DIRY chunk (an integral number of words).
DataLength	The size of the Data (an integral number of words).
Data	

where:

ChunkIndex
: is a word containing the zero-origin index within the chunk file header of the corresponding LIB_DATA chunk. Conventionally, the first three chunks of an OFL file are LIB_DIRY, LIB_TIME and LIB_VRSN, so ChunkIndex is at least 3. A ChunkIndex of 0 means the directory entry is unused.

  The corresponding LIB_DATA chunk entry gives the offset and size of the library module in the library file.

EntryLength
: is a word containing the number of bytes in this LIB_DIRY entry, always a multiple of 4.

DataLength
: is a word containing the number of bytes used in the data section of this LIB_DIRY entry, also a multiple of 4.

Data
: consists of, in order:

  - a zero-terminated string (the name of the library member). Strings should contain only ISO-8859 non-control characters (codes [0-31], 127 and 128+[0-31] are excluded). The string field is the name used to identify this library module. Typically it is the name of the file from which the library member was created.
  - any other information relevant to the library module (often empty).

- a two-word, word-aligned time stamp. The format of the time stamp is described in *Time stamps* on page 14-7. Its value is an encoded version of the last-modified time of the file from which the library member was created.

14.3.3 LIB_VRSN

The version chunk contains a single word whose value is 1.

14.3.4 LIB_DATA

A `LIB_DATA` chunk contains one of the library members indexed by the `LIB_DIRY` chunk. The endianness or byte order of this data is, by assumption, the same as the byte order of the containing library/chunk file.

No other interpretation is placed on the contents of a member by the library management tools. A member could itself be a file in chunk file format or even another library.

14.4 Time stamps

A library time stamp is a pair of words that encode:

- a six byte count of centiseconds since 00:00:00 1st January 1900
- a two byte count of microseconds since the last centisecond.

First (most significant) word

Contains the most significant 4 bytes of the 6 byte centisecond count.

Second (least significant) word

Contains the least significant two bytes of the six byte centisecond count in the most significant half of the word and the two byte count of microseconds since the last centisecond in the least significant half of the word. This is usually 0.

Time stamp words are stored in target system byte order. They must have the same endianness as the containing chunk file.

14.4.1 LIB_TIME

The `LIB_TIME` chunk contains a two-word (eight-byte) time stamp recording when the library was last modified.

14.5 Object code libraries

An object code library is a library file whose members are files in ARM Object Format. An object code library contains two additional chunks:

- an external symbol table chunk named `OFL_SYMT`
- a time stamp chunk named `OFL_TIME.`

14.5.1 OFL_SYMT

The external symbol table contains an entry for each external symbol defined by members of the library, together with the index of the chunk containing the member defining that symbol.

The `OFL_SYMT` chunk has exactly the same format as the `LIB_DIRY` chunk except that the Data section of each entry contains only a string, the name of an external symbol, and between one and four bytes of NULL padding, as follows:

Table 14-3 OFL_SYMT chunk layout

ChunkIndex	
EntryLength	The size of this `OFL_SYMT` chunk (an integral number of words).
DataLength	The size of the External Symbol Name and Padding (an integral number of words).
External Symbol Name	
Padding	

`OFL_SYMT` entries do not contain time stamps.

14.5.2 OFL_TIME

The `OFL_TIME` chunk records when the `OFL_SYMT` chunk was last modified and has the same format as the `LIB_TIME` chunk (see *Time stamps* on page 14-7).

Chapter 15
ARM Object Format

This chapter describes the ARM Object Format. It contains the following sections:

15.1 ARM Object Format

This section describes the ARM Object Format (AOF).

The following terms apply throughout this section:

object file refers to a file in ARM Object Format.

address for data in a file, this means *offset from the start of the file.*

15.1.1 Areas

An object file written in AOF consists of any number of named, attributed areas. Attributes include:

- read-only
- reentrant
- code
- data
- position-independent.

For details see *Attributes and alignment* on page 15-9.

Typically, a compiled AOF file contains a read-only code area, and a read-write data area (a zero-initialized data area is also common, and reentrant code uses a separate based area for address constants).

15.1.2 Relocation directives

Associated with each area is a (possibly empty) list of relocation directives which describe locations that the linker will have to update when:

- a non-zero base address is assigned to the area
- a symbolic reference is resolved.

Each relocation directive may be given relative to the (not yet assigned) base address of an area in the same AOF file, or relative to a symbol in the symbol table. Each symbol may:

- have a definition within its containing object file which is local to the object file
- have a definition within the object file which is visible globally (to all object files in the link step)
- be a reference to a symbol defined in some other object file.

15.1.3 Byte sex or endianness

An AOF file can be produced in either little-endian or big-endian format.

There is no guarantee that the endianness of an AOF file will be the same as the endianness of the system used to process it (the endianness of the file is always the same as the endianness of the target ARM system).

15.1.4 Alignment

Strings and bytes may be aligned on any byte boundary. AOF fields defined in this document make no use of halfwords and align words on 4-byte boundaries.

Within the contents of an AOF file, the alignment of words and halfwords is defined by the use to which AOF is being put. For all current ARM-based systems, words are aligned on 4-byte boundaries and halfwords on 2-byte boundaries.

15.2 Overall structure of an AOF file

An AOF file contains a number of separate pieces of data. To simplify access to the data, and to give a degree of extensibility to tools which process AOF, the object file format is itself layered on another format called *Chunk File Format*, which provides a simple and efficient means of accessing and updating distinct chunks of data within a single file.

15.2.1 Chunk file format

A file written in chunk file format consists of a header, and one or more chunks. The header is always positioned at the beginning of the file. A chunk is accessed through the header. The header contains the number, size, location, and identity of each chunk in the file.

The size of the header may vary between different chunk files, but it is fixed for each file. Not all entries in a header need be used, thus limited expansion of the number of chunks is permitted without a wholesale copy. A chunk file can be copied without knowledge of the contents of its chunks.

Chunk file header

The chunk file header consists of two parts:

- the first part is a fixed length part of three words
- the second part contains a four word entry for each chunk in the file.

The first part of the header contains the following three word sized fields:

`ChunkFileId` Marks the file as a chunk file. Its value is 0xC3CBC6C5. The endianness of the chunk file can be determined from this value (if it appears to be 0xC5C6CBC3 when read as a word, each word value must be byte-reversed before use).

`max_chunks` Defines the number of the entries in the header, fixed when the file is created.

`num_chunks` Defines how many chunks are currently used in the file, which can vary from 0 to `max_chunks`. It is redundant in that it can be found by scanning the entries.

The second part of the header contains a four word entry for each chunk in the file. The number of entries is given by the `num_chunks` field in the first part of the header.

`chunkId` Is an 8-byte field identifying what data the chunk contains. Note that this is an 8-byte field, *not* a 2-word field, so it has the same byte order independent of endianness.

`file_offset` Is a one-word field defining the byte offset within the file of the start of the chunk. All chunks are word-aligned, so it must be divisible by four. A value of zero indicates that the chunk entry is unused.

`size` Is a one-word field defining the exact byte size of the chunk's contents (which need not be a multiple of four).

Identifying data types

The `chunkId` field provides a conventional way of identifying what type of data a chunk contains. It has eight characters, and is split into two parts:

- the first four characters contain a unique name allocated by a central authority
- the remaining four characters can be used to identify component chunks within this domain.

The eight characters are stored in ascending address order, as if they formed part of a NULL-terminated string, independent of endianness.

For AOF files, the first part of each chunk name is `OBJ_`. The second components are defined in the following section.

15.2.2 ARM object format

Each piece of an object file is stored in a separate, identifiable chunk. AOF defines five chunks as shown in Table 15-1.

Table 15-1 AOF chunks

Chunk	Chunk name
AOF Header	`OBJ_HEAD`
Areas	`OBJ_AREA`
Identification	`OBJ_IDFN`
Symbol Table	`OBJ_SYMT`
String Table	`OBJ_STRT`

Only the `AOF Header` and `AREAS` chunks must be present, but a typical object file contains all five of the above chunks.

Each name in an object file is encoded as an offset into the string table, stored in the `OBJ_STRT` chunk *The String Table Chunk (OBJ_STRT)* on page 15-21. This allows the variable-length nature of names to be factored out from primary data formats.

A feature of ARM Object Format is that chunks may appear in any order in the file (for example, the ARM C compiler and the ARM assembler produce their AOF chunks in different orders).

A language translator or other utility may add additional chunks to an object file, for example, a language-specific symbol table or language-specific debugging data. Therefore it is conventional to allow space in the chunk header for additional chunks. Space for eight chunks is conventional when the AOF file is produced by a language processor which generates all five chunks described here.

Note

The AOF header chunk should not be confused with the chunk file header.

15.3 The AOF header chunk (OBJ_HEAD)

The AOF header consists of two contiguous parts:

- the first part is a fixed size part of six words that describes the contents and nature of the object file.
- the second part has a variable length (specified in the first part of the header), and consists of a sequence of *area headers* describing the areas within the `OBJ_AREA` chunk.

Part one contains the following word sized fields:

`Object File Type`
: The value 0xC5E2D080 marks the file as being in *relocatable object format* (the usual output of compilers and assemblers and the usual input to the linker). The endianness of the object code can be deduced from this value and must be identical to the endianness of the containing chunk file.

`Version Id`
: Encodes the AOF version number. The current version number is 310 (0x136).

`Number of Areas`
: The code and data of an object file are encapsulated in a number of separate areas in the `OBJ_AREA` chunk, each with a name and some attributes (see *Attributes and alignment* on page 15-9).

  Each area is described in the variable-length part of the AOF header which immediately follows the fixed part. `Number_of_Areas` gives the number of areas in the file and, equivalently, the number of `AREA` declarations that follow the fixed part of the AOF header.

`Number of Symbols`
: If the object file contains a symbol table chunk (named `OBJ_SYMT`), `Number of Symbols` records the number of symbols in the symbol table.

  One of the areas in an object file may be designated as containing the start address of any program which is linked to include the file. If this is the case, the entry address is specified as an `Entry Area Index`, `Entry Offset` pair.

`Entry Area Index`
: `Entry Area Index`, in the range 1 to `Number of Areas`, gives the 1-origin index in the following array of area headers of the area containing the entry point.

A value of 0 for `Entry Area Index` signifies that no program entry address is defined by this AOF file.

`Entry Offset`

The entry address is defined to be the base address of the entry area plus `Entry Offset`.

Part two of the AOF header consists of a sequence of area headers. Each area header is five words long, and contains the following word length fields:

`Area Name` Gives the offset of that name in the string table (stored in the `OBJ_STRT` chunk. Each area within an object file must be given a unique name. See *The String Table Chunk (OBJ_STRT)* on page 15-21.

`Attributes and Alignment`

This word contains bit flags that specify the attributes and alignment of the area. The details are given in *Alignment* on page 15-3.

`Area Size` Gives the size of the area in bytes. This value must be a multiple of 4. Unless the `Not Initialised bit` (bit 12) is set in the area attributes (see *Attributes and alignment* on page 15-9), there must be this number of bytes for this area in the `OBJ_AREA` chunk. If the `Not Initialised` bit is set, there must be no initializing bytes for this area in the `OBJ_AREA` chunk.

`Number of Relocations`

Specifies the number of relocation directives that apply to this area (which is equivalent to the number of relocation records following the contents of the area in the `OBJ_AREA` chunk. See *The AREAS chunk (OBJ_AREA)* on page 15-13).

`Base Address`

Is unused unless the area has the absolute attribute. In this case, the field records the base address of the area. In general, giving an area a base address prior to linking will cause problems for the linker and may prevent linking altogether, unless only a single object file is involved.

An unused Base Address is denoted by the value 0.

15.3.1 Attributes and alignment

Each area has a set of attributes encoded in the most significant 24 bits of the `Attributes + Alignment` word. The least significant eight bits of this word encode the alignment of the start of the area as a power of 2 and must have a value between 2 and 32 (this value denotes that the area should start at an address divisible by $2^{alignment}$). Table 15-2 gives a summary of the attributes.

Table 15-2 Area attributes summary

Bit	Mask	Attribute Description
8	0x00000100	Absolute attribute
9	0x00000200	Code attribute
10	0x00000400	Common block definition
11	0x00000800	Common block reference
12	0x00001000	Uninitialized (zero-initialized)
13	0x00002000	Read-only
14	0x00004000	Position independent
15	0x00008000	Debugging tables
		Code areas only
16	0x00010000	Complies with the 32-bit APCS
17	0x00020000	reentrant code
18	0x00040000	Uses extended FP instruction set
19	0x00080000	No software stack checking
20	0x00100000	All relocations are of Thumb code
21	0x00200000	Area may contain ARM halfword instructions
22	0x00400000	Area suitable for ARM/Thumb interworking

Some combinations of attributes are meaningless, for example, read-only and zero-initialized.

The linker orders areas in a generated image in the following order:

- by attributes
- by the (case-significant) lexicographic order of area names
- by position of the containing object module in the link list.

The position in the link list of an object module loaded from a library is not predictable. The precise significance to the linker of area attributes depends on the output being generated.

Bit 8 Encodes the *absolute* attribute and denotes that the area must be placed at its *Base Address*. This bit is not usually set by language processors.

Bit 9 Encodes the *code* attribute:

- **1** Indicates code in the area.
- **0** Indicates data in the area.

Bit 10 Specifies that the area is a common definition.

Common areas with the same name are overlaid on each other by the linker. The `Area Size` field of a common definition area defines the size of a common block. All other references to this common block must specify a size which is smaller than or equal to the definition size.

If, in a link step, there is more than one definition of an area with the *common definition* attribute (area of the given name with bit 10 set), each of these areas must have exactly the same contents. If there is no definition of a common area, its size will be the size of the largest common reference to it.

Although common areas conventionally hold data, you can use bit 10 in conjunction with bit 9 to define a common block containing code. This is useful for defining a code area which must be generated in several compilation units, but which should be included in the final image only once.

Bit 11 Defines the area to be a reference to a common block, and precludes the area having initializing data (see *Bit 12*). In effect, bit 11 implies bit 12. If both bits 10 and 11 are set, bit 11 is ignored.

Bit 12 Encodes the *zero-initialized* attribute, specifying that the area has no initializing data in this object file, and that the area contents are missing from the `OBJ_AREA` chunk.

Typically, this attribute is given to large municipalized data areas. When a municipalized area is included in an image, the linker either includes a read-write area of binary zeros of appropriate size, or maps a read-write area of appropriate size that will be zeroed at image startup time. This attribute is incompatible with the read-only attribute (see Bit 13, below).

Whether or not a zero-initialized area is re-zeroed if the image is re-entered is a property of the relevant image format and/or the system on which it will be executed. The definition of AOF neither requires nor precludes re-zeroing.

A combination of bit 10 (common definition) and bit 12 (zero-initialized) has exactly the same meaning as bit 11 (reference to common).

Bit 13 Encodes the *read only* attribute and denotes that the area will not be modified following relocation by the linker. The linker groups read-only areas together so that they may be write-protected at runtime, hardware permitting. Code areas and debugging tables must have this bit set. The setting of this bit is incompatible with the setting of bit 12.

Bit 14 Encodes the *position independent* (PI) attribute, usually only of significance for code areas. Any reference to a memory address from a PI area must be in the form of a link-time-fixed offset from a base register (for example, a pc-relative branch offset).

Bit 15 Encodes the *debugging table* attribute and denotes that the area contains symbolic debugging tables. The linker groups these areas together so they can be accessed as a single continuous chunk at or before runtime (usually, a debugger extracts its debugging tables from the image file prior to starting the debuggee). Usually, debugging tables are read-only and, therefore, have bit 13 set also. In debugging table areas, bit 9 (the *code* attribute) is ignored.

Bits 16-22 encode additional attributes of code areas and must be non-zero only if the area has the code attribute (bit 9) set. Bits 20-22 can be non-zero for data areas.

Bit 16 Encodes the *32-bit PC attribute*, and denotes that code in this area complies with a 32-bit variant of the APCS.

Bit 17 Encodes the *reentrant* attribute, and denotes that code in this area complies with a reentrant variant of the APCS.

Bit 18 When set, denotes that code in this area uses the ARM floating-point instruction set. Specifically, function entry and exit use the `LFM` and `SFM` floating-point save and restore instructions rather than multiple `LDFE`s and `STFE`s. Code with this attribute may not execute on older ARM-based systems.

Bit 19 Encodes the *No Software Stack Check* attribute, denoting that code in this area complies with a variant of the APCS without software stack-limit checking.

Bit 20 Indicates that this area is a Thumb code area.

Bit 21 Indicates that this area may contain ARM halfword instructions. This bit is set by armcc when compiling code for a processor with halfword instructions such as the ARM7TDMI.

Bit 22 Indicates that this area has been compiled to be suitable for ARM/Thumb interworking. See the *ARM Software Development Toolkit User Guide*.

Bits 23 to 31 Are reserved and are set to 0.

15.4 The AREAS chunk (OBJ_AREA)

The AREAs chunk contains the actual area contents, such as code, data, debugging data, together with their associated relocation data. An area is simply a sequence of bytes. The endianness of the words and halfwords within it must agree with that of the containing AOF file. An area layout is:

```
Area 1
Area 1 Relocation
...
Area n
Area n Relocation
```

An area is followed by its associated table of relocation directives (if any). An area is either completely initialized by the values from the file or is initialized to zero, as specified by bit 12 of its area attributes. Both area contents and table of relocation directives are aligned to 4-byte boundaries.

15.5 Relocation directives

A relocation directive describes a value which is computed at link time or load time, but which cannot be fixed when the object module is created.

In the absence of applicable relocation directives, the value of a byte, halfword, word or instruction from the preceding area is exactly the value that will appear in the final image.

A field may be subject to more than one relocation.

Figure 15-1 shows a relocation directive.

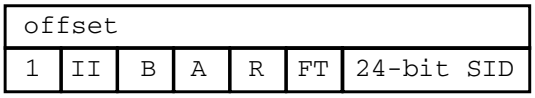

Figure 15-1 Relocation directive

`Offset` is the byte offset in the preceding area of the subject field to be relocated by a value calculated as described below.

The interpretation of the 24-bit `SID` field depends on the value of the `A` bit (bit 27):

A=1 The subject field is relocated (as further described below) by the value of the symbol of which `SID` is the zero-origin index in the symbol table chunk.

A=0 The subject field is relocated (as further described below) by the base of the area of which `SID` is the zero-origin index in the array of areas, (or, equivalently, in the array of area headers).

The two-bit field type `FT` (bits 25, 24) describes the subject field:

00 the field to be relocated is a byte.

01 the field to be relocated is a halfword (two bytes).

10 the field to be relocated is a word (four bytes).

11 the field to be relocated is an instruction or instruction sequence.

If bit 0 of the relocation offset is set, this identifies a Thumb instruction sequence, otherwise it is taken to be an ARM instruction sequence.

Bytes, halfwords and instructions may only be relocated by values of small size. Overflow is faulted by the linker.

An ARM branch or branch-with-link instruction is always a suitable subject for a relocation directive of field type *instruction*. For details of other relocatable instruction sequences, refer to 3.6 Handling Relocation Directives on page 3-16.

If the subject field is an instruction sequence, the address in `Offset` points to the first instruction of the sequence, and the `II` field (bits 29 and 30) constrains how many instructions may be modified by this directive:

00 no constraint (the linker may modify as many contiguous instructions as it needs to).

01 the linker will modify at most 1 instruction.

10 the linker will modify at most 2 instructions.

11 the linker will modify at most 3 instructions.

The `R` (pc-relative) bit, modified by the `B` (based) bit, determines how the relocation value is used to modify the subject field:

R (bit 26) = 0 and B (bit 28) = 0

This specifies plain additive relocation. The relocation value is added to the subject field. In pseudo code:

```
subject_field = subject_field + relocation_value
```

R (bit 26) = 1 and B (bit 28) = 0

This specifies pc-relative relocation. To the subject field is added the difference between the relocation value and the base of the area containing the subject field. In pseudo code:

```
subject_field =
subject_field +
(relocation_value-base_of_area_containing(subject_fie
ld))
```

As a special case, if `A` is 0, and the relocation value is specified as the base of the area containing the subject field, it is not added and:

```
subject_field =
subject_field - base_of_area_containing(subject_field)
```

This caters for relocatable pc-relative branches to fixed target addresses.

If `R` is 1, `B` is usually 0. A `B` value of 1 is used to denote that the inter-link-unit value of a branch destination is to be used, rather than the more usual intra-link-unit value.

R (bit 26) = 0 and B (bit 28) = 1

This specifies based area relocation. The relocation value must be an address within a based data area. The subject field is incremented by the difference between this value and the base address of the consolidated based area group (the linker consolidates all areas based on the same base register into a single, contiguous region of the output image).

In pseudo code:

```
subject_field =
subject_field +
(relocation_value -
base_of_area_group_containing(relocation_value))
```

For example, when generating reentrant code, the C compiler places address constants in an address constant area based on register sb, and loads them using sb-relative `LDR` instructions. At link time, separate address constant areas will be merged and sb will no longer point where presumed at compile time. B type relocation of the `LDR` instructions corrects for this.

Bits 29 and 30 of the relocation flags word must be 0. Bit 31 must be 1.

15.6 Symbol Table Chunk Format (OBJ_SYMT)

The `Number of Symbols` field in the fixed part of the AOF header (`OBJ_HEAD` chunk) defines how many entries there are in the symbol table. Each symbol table entry is four words long and contains the following word length fields:

`Name` Is the offset in the string table (in chunk `OBJ_STRT`) of the character string name of the symbol.

`Attributes`
Are summarized in Table 15-2. Refer to *Symbol attributes* on page 15-18 for a full description of the attributes.

`Value` Is meaningful only if the symbol is a defining occurrence (bit 0 of `Attributes` set), or a common symbol (bit 6 of `Attributes` set):

- if the symbol is *absolute* (bits 0-2 of `Attributes` set), this field contains the value of the symbol
- if the symbol is a common symbol (bit 6 of `Attributes` set), this contains the byte length of the referenced common area.
- otherwise, `Value` is interpreted as an offset from the base address of the area named by `Area Name`, which must be an area defined in this object file.

`Area Name` Is meaningful only if the symbol is a non-absolute defining occurrence (bit 0 of `Attributes` set, bit 2 unset). In this case it gives the index into the string table for the name of the area in which the symbol is defined (which must be an area in this object file).

15.6.1 Symbol attributes

Table 15-3 summarizes the symbol attributes.

Table 15-3 Symbol attributes

Bit	Mask	Attribute description
0	0x00000001	Symbol is defined in this file
1	0x00000002	Symbol has a global scope
2	0x00000004	Absolute attribute
3	0x00000008	Case-insensitive attribute
4	0x00000010	Weak attribute
6	0x00000040	Common attribute
		Code symbols only:
8	0x00000100	Code area datum attribute
9	0x00000200	FP args in FP regs attribute
12	0x00001000	Thumb symbol

The Symbol `Attributes` word is interpreted as follows:

Bit 0 Denotes that the symbol is defined in this object file.

Bit 1 Denotes that the symbol has global scope and can be matched by the linker to a similarly named symbol from another object file.

- **01** Bit 1 unset, bit 0 set. Denotes that the symbol is defined in this object file and has scope limited to this object file (when resolving symbol references, the linker will only match this symbol to references from within the same object file).
- **10** Bit 1 set, bit 0 unset. Denotes that the symbol is a reference to a symbol defined in another object file. If no defining instance of the symbol is found, the linker attempts to match the name of the symbol to the names of common blocks. If a match is found, it is as if an identically-named symbol of global scope were defined, taking its value from the base address of the common area.
- **11** Denotes that the symbol is defined in this object file with global scope (when attempting to resolve unresolved references, the linker will match this definition to a reference from another object file).
- **00** Is reserved.

Bit 2 Encodes the *absolute* attribute which is meaningful only if the symbol is a defining occurrence (bit 0 set). If set, it denotes that the symbol has an absolute value, for example, a constant. If unset, the symbol value is relative to the base address of the area defined by the *Area Name* field of the symbol.

Bit 3 Encodes the *case insensitive reference* attribute which is meaningful only if the symbol is an external reference (bits 1, 0 = 10). If set, the linker will ignore the case of the symbol names it tries to match when attempting to resolve this reference.

Bit 4 Encodes the *weak* attribute which is meaningful only if the symbol is an external reference (bits 1, 0 = 10). It denotes that it is acceptable for the reference to remain unsatisfied and for any fields relocated via it to remain unrelocated. The linker ignores weak references when deciding which members to load from an object library.

Bit 5 Is reserved and must be set to 0.

Bit 6 Encodes the *common* attribute, which is meaningful only if the symbol is an external reference (bits 1, 0 = 10). If set, the symbol is a reference to a common area with the symbol's name. The length of the common area is given by the symbol's *Value* field (see above). The linker treats common symbols much as it treats areas having the *Common Reference* attribute. All symbols with the same name are assigned the same base address, and the length allocated is the maximum of all specified lengths.

If the name of a common symbol matches the name of a common area, these are merged and the symbol identifies the base of the area.

All common symbols for which there is no matching common area (reference or definition) are collected into an anonymous, linker-created, pseudo-area.

Bit 7 Is reserved and must be set to 0.

Bits 8-11 encode additional attributes of symbols defined in code areas.

Bit 8 Encodes the *code datum* attribute which is meaningful only if this symbol defines a location within an area having the *Code* attribute. It denotes that the symbol identifies a (usually read-only) datum, rather than an executable instruction.

Bit 9 Encodes the *floating-point arguments in floating-point registers* attribute. This is meaningful only if the symbol identifies a function entry point. A symbolic reference with this attribute cannot be matched by the linker to a symbol definition which lacks the attribute.

Bit 10 Is reserved and must be set to 0.

Bit 11 Is reserved and must be set to 0.

Bit 12 The Thumb attribute. This is set if the symbol is a Thumb symbol.

15.7 The String Table Chunk (OBJ_STRT)

The string table chunk contains all the print names referred to from the *header* and *symbol table* chunks. This separation is made to factor out the variable length characteristic of print names from the key data structures.

A print name is stored in the string table as a sequence of non-control characters (codes 32-126 and 160-255) terminated by a NULL (0) byte, and is identified by an offset from the start of the table. The first four bytes of the string table contain its length (including the length of its length word), so no valid offset into the table is less than four, and no table has length less than four.

The endianness of the length word must be identical to the endianness of the AOF and chunk files containing it.

15.8 The Identification Chunk (OBJ_IDFN)

This chunk should contain a string of printable characters (codes 10-13 and 32-126) terminated by a NULL (0) byte, which gives information about the name and version of the tool which generated the object file.

Use of codes in the range 128-255 is discouraged, as the interpretation of these values is host-dependent.