ABI for the ARM Architecture: Support for Debugging Overlaid Programs

Document number:	ARM IHI 0049A, current through ABI release 2.08
Date of Issue:	10th October 2008, reissued 28th October 2009

Abstract

This specification defines an extension to the *ABI for the ARM Architecture* to support debugging overlaid programs. No tool chain is required to support this extension but tools that support debugging overlaid programs should do so in one of the ways specified in §3.

Keywords

Debugging ABI for the ARM Architecture; debugging; ABI

How to find the latest release of this specification or report a defect in it

Please check the *ARM Information Center* (http://infocenter.arm.com/) for a later release if your copy is more than one year old (navigate to the *Software Development Tools* section, *Application Binary Interface for the ARM Architecture* subsection).

Please report defects in this specification to *arm* dot *eabi* at *arm* dot *com*.

Contents

1 ABOUT THIS DOCUMENT

1.1 Change control

1.1.1 Current status and anticipated changes

1.1.2 Change history

Issue	Date	By	Change
A	10th October 2008	LS	First public release.

1.2 References

This document refers to the following documents.

Ref	Author(s) or links	Title
ABI	http://infocenter.arm.com/help/index.jsp → *Software development tools →* *Application Binary Interface for the ARM Architecture*	Application Binary Interface for the ARM® Architecture
ABI Addenda	*Ibid → Addenda to, and Errata in, the ABI for the ARM Architecture*	Addenda to, and Errata in, the ABI for the ARM Architecture
AADWARF	*Ibid → DWARF for the ARM Architecture*	DWARF for the ARM Architecture
AAELF	*Ibid → ELF for the ARM Architecture*	ELF for the ARM Architecture
GNU OV	http://sourceware.org/gdb/current/onlinedocs/gdb_12.html#SEC100	Debugging Programs That Use Overlays (GDB documentation suite)

1.3 Terms and abbreviations

This document defines its terms and abbreviations in the document text.

1.4 Your licence to use this specification

IMPORTANT: THIS IS A LEGAL AGREEMENT ("LICENCE") BETWEEN YOU (AN INDIVIDUAL OR SINGLE ENTITY WHO IS RECEIVING THIS DOCUMENT DIRECTLY FROM ARM LIMITED) ("LICENSEE") AND ARM LIMITED ("ARM") FOR THE SPECIFICATION DEFINED IMMEDITATELY BELOW. BY DOWNLOADING OR OTHERWISE USING IT, YOU AGREE TO BE BOUND BY ALL OF THE TERMS OF THIS LICENCE. IF YOU DO NOT AGREE TO THIS, DO NOT DOWNLOAD OR USE THIS SPECIFICATION.

"Specification" means, and is limited to, the version of the specification for the Applications Binary Interface for the ARM Architecture comprised in this document. Notwithstanding the foregoing, "Specification" shall not include (i) the implementation of other published specifications referenced in this Specification; (ii) any enabling technologies that may be necessary to make or use any product or portion thereof that complies with this Specification, but are not themselves expressly set forth in this Specification (e.g. compiler front ends, code generators, back ends, libraries or other compiler, assembler or linker technologies; validation or debug software or hardware; applications, operating system or driver software; RISC architecture; processor microarchitecture); (iii) maskworks

1.5 Acknowledgements

Lauterbach Datentechnik GmbH gave valuable review of earlier drafts of this specification.

2 THE INTERFACE BETWEEN LINKERS AND DEBUGGERS

2.1 Summary

The *ABI for the ARM® Architecture* [ABI] specifies ELF [AAELF] as the executable file format and DWARF 3.0 [AADWARF] as the debugging data format.

This note describes the obligations a producer of an executable ELF file (a linker) must meet to support debugging overlaid programs.

2.2 Terminology

In this note the terms *virtual address* and *address* are used interchangeably to describe addresses in a target system used by a program. This note is not concerned with the possibility that an external agent such as a debugger might 'see' addresses differently to an executing program.

A linker has two views of a program's address space that become distinct in the presence of *overlaid*, *position-independent*, and *relocatable* program fragments (code or data).

- The *load address* of a program fragment is the target address to a linker expects an external agent such as a program loader, dynamic linker, or debugger to copy the fragment from the ELF file. This is not necessarily the address at which the fragment will execute.
- The *execution address* of a program fragment is the target address at which a linker expects the fragment will reside whenever it participates in the program's execution.

Of course, if a fragment is position-independent or relocatable, its execution address can vary during execution.

2.3 Standard ELF views

The ELF standard specifies two *views* of an executable ELF file.

In the *program view*, each *program header* of type PT_LOAD describes:

- A contiguous region of the file containing the initializing content for a *program segment*.
- A contiguous region of target address-space to which an external agent will copy that content.

In a program header, target addresses *should* be *load addresses* (§2.2) because an external agent is expected to load the program segment there.

In the *section view*, each *section header* describes:

- A contiguous region of the file occupied by the content of the section.
- And, if the section will appear in memory, a corresponding contiguous region of the target address-space. These addresses *must* be *execution addresses*.

The ELF standard permits the section view to be omitted from an executable ELF file and this is typically done when executable files are not intended to be debugged. The segment view suffices to support loading and execution.

In practice, the section view is *never* omitted when an ELF file is intended to be debugged.

DWARF debug tables and the section view of an ELF file can embody only one interpretation of target addresses. Because debuggers debug the *execution* of a program it is *logically necessary* for this to be the *execution address* view. By the same argument, ELF symbols must (almost always) define target execution addresses.

2.4 Relating different views of target addresses

In the absence of *relocatable*, *position-independent*, or *overlaid* program fragments, a debugger has no use for load addresses.

For example, a debugger stepping through a self-installing program will always ‘see’ execution addresses.

Load addresses might still have meaning to the user of a debugger, but their availability can be a *quality of implementation*. Non availability does not reduce a debugger’s *necessary* functionality.

Relocatable and *position-independent* program fragments cause difficulties for debuggers that are beyond the scope of this note so we mention them no more.

Overlaid program fragments cause the following difficulty.

> Multiple debug sections that should refer to distinct program fragments (and that *do* refer to distinct relocatable program fragments prior to static linking) actually refer to the same region of target memory that is time-multiplexed between multiple program fragments.

Stated simply, given a target execution address, several different debug sections might relate to it and there is no obvious way to choose among them.

The remainder of this section explains how to make the relationship between target addresses and debug sections unambiguous.

2.4.1 Finding section and symbol load addresses

Each *program header* PH of type PT_LOAD defines

- A half-open extent of the ELF file, [PH.p_offset, PH.p_offset + PH.p_filesz).
- A half-open extent of load-address space, [PH.p_paddr, PH.p_paddr + PH.p_memsz).

It is guaranteed that p_memsz ≥ p_filesz.

Note Strictly speaking the ELF standard guarantees that the memory interval [PH.p_vaddr + PH.p_filesz, PH.p_vaddr + PH.p_memsz) will be set to zero. Many embedded systems allow it to be uninitialized.

Note Some linkers – notably GNU ld – use PH.p_**paddr** to hold the load address of a segment. We adopt that convention in this note and propose it as an extension to the ABI in §3.

Each *section header* SH defines:

- A half-open extent of the ELF file, [SH.sh_offset, SH.sh_offset + *filesz*), where *filesz* is SH.sh_size or 0 if SH.sh_type = SHT_NOBITS.
- A half-open extent of execution-address space, [SH.sh_addr, SH.sh_addr + SH.sh_size).

For any section SH whose file extent overlaps the file extent of a segment PH and any file offset *off* that lies in *both* file extents the load address LA and execution address EA corresponding to *off* are:

> LA(*off*) = PH.p_paddr + (*off* - PH.p_offset)
>
> EA(*off*) = SH.sh_addr + (*off* - SH.sh_offset)

Conditional on the corresponding file offset *off* lying in *both* the segment file extent *and* the section file extent

> LA = EA + PH.p_paddr - SH.sh_addr + SH.sh_offset - PH.p_offset

This gives the load address corresponding to each target execution address and, in the presence of overlaid program fragments will give multiple load addresses for the same execution address.

In particular, this allows the load address of every section that is part of the program to be computed from information already present in the ELF file.

Note Normally a program section cannot intersect more than one program segment.

Note When two or more segments are overlaid at the same *load address* and contain only sections of type SHT_NOBITS (zero-initialized or uninitialized data) and there is no intervening file content between the segments, the sections and the segments all have identical (empty) file extents. It is then impossible to match sections to a loaded segment via a unique file extent which makes it impossible to locate the debugging sections appropriate to the loaded segment.

This obscure corner case can be avoided if a linker ensures that every program segment has a unique file offset, p_offset. This can be done by adding padding bytes between adjacent segments with empty file extents (§3.2).

Once a load address is known for each section, the load address of every section-relative symbol S can be found.

S.st_shndx identifies the section header SH for the section in which S is defined.

S.st_value - SH.sh_addr is the offset of S in the section described by SH.

S.load_address = SH.load_address + (S.st_value - SH.sh_addr).

From above:

SH.load_address = SH.sh_addr + PH.p_paddr - SH.sh_addr + SH.sh_offset - PH.p_offset
= PH.p_paddr + (SH.sh_offset - PH.p_offset)

2.4.2 Finding which overlay is currently executing

In a typical embedded application, each section S in a set {S} of sections with overlapping execution extents has a distinct extent in load-address space. The section executing is the one for which the content of the execution-address space extent is identical to the content in the corresponding load-address space extent[1].

Note This definition only works for read-only segments that have not been accidentally corrupted. In other cases a debugger must observe or collude with the overlay manager to discover which segment is live.

Note If the overlay system uses a centralized overlay manager (rather than loading overlays in an ad-hoc, distributed manner) it might be possible for a debugger to observe the load address and execution address used by the overlay manager in a code fragment resembling

`memcpy(`*execution address*, *load address*, *segment length*`)`

The static structure of overlays is, of course, discernable from *execution address*, *load address*, and *section length* of each section that overlaps another in the execution-address space.

2.4.3 Relating debug sections to program sections

In a relocatable file, a debug section refers to a location in a program section via a relocated location.

A relocation directive refers to the debug section being relocated via the sh_info field in the relocation section header and the r_offset field in the relocation itself. It refers to the program section via a symbol (identified by ELF32_R_SYM(r_info)) that refers to the program section via st_shndx and st_value (an offset in the section).

At this stage of linking, a reference from a debug section to a location in a program section is a pair of pairs

<debug section index, debug section offset>, <program section index, program section offset>

During static linking the *program* pair is reduced to single value, the *execution address*. This is ambiguous in the presence of overlaid sections.

Resolving the ambiguity requires some of the original relocation information. We propose two ways to represent that in an ELF file.

[1] Assuming the program has not altered writable memory and that initializing contents are unique.

- Retain the relevant subset (or all) of the original relocations in the executable ELF file.
- Emit a new ELF section called `.ARM.debug_overlay` of type SHT_ARM_DEBUG_OVERLAY = SHT_LOUSER + 4 containing a table of entries as follows:

  debug section offset, debug section index, program section index

The description earlier in this section shows that the second representation can be calculated from the relevant subset of the retained relocation data.

GNU *ld* has an option (`--emit-relocs`) to retain all relocations in the executable file. Clearly this is sufficient.

A better option is to retain only relocations of debug sections (those with names matching `*debug*`) with respect to overlaid program sections (`--emit-overlay-debug-relocs`). An overlay-aware linker will readily recognize these sections.

For some linkers it might be easier to build a `.ARM.debug_overlay` section directly, as each relocation directive is processed, than to emit the original relocations filtered for relevance.

3 THE ABI EXTENSION

We extend the *ABI for the ARM Architecture* (ABI) as noted in this section. The extension is optional and no tool chain is required to support in order to claim conformance to the ABI. However, tools that support debugging overlaid programs should do so in one of the ways specified here.

3.1 Terminology

A linker has two views of a program's address space that become distinct in the presence of *overlaid* program fragments (code or data).

- The *load address* of a program fragment is the address to which a linker expects an external agent such as a program loader, dynamic linker, or debugger to copy the fragment from the ELF file. This is not necessarily the address at which the fragment will execute.
- The *execution address* of a program fragment is the address at which a linker expects the fragment will reside whenever it participates in the program's execution.

3.2 Linker obligations

A linker claiming to support the debugging of overlaid programs shall ensure the following in the executable ELF files it produces.

- Each program fragment that overlaps another in the execution address space shall be described by a distinct ELF section header.
- Target addresses recorded in section header sh_addr fields and symbol st_value fields shall be *execution addresses*.
- Target addresses recorded in p_paddr fields of program headers of type `PT_LOAD` shall be *load addresses.*
- Each program segment described by a program header PH of type `PT_LOAD` shall occupy a different extent [PH.p_offset, PH.p_offset + PH.p_filesz) in the ELF file. (An empty extent shall not overlap any other extent).

In addition, a linker claiming to support debugging of overlaid programs shall do *at least one* of the following.

- Provide a means to retain all original relocations in the executable file.
  GNU *ld* does this using the command option `--emit-relocs`.
- Provide a means to retain just those original relocations that relocate debug sections with respect to overlaid program sections. A linker might provide a command option such as `--emit-overlay-debug-relocs`.
- Add a *debug-overlay* ELF section (specified in §3.2.1, below) to the executable file.

3.2.1 The debug-overlay section

Table 1, The debug-overlay section header

Field	Value
sh_name	`.ARM.debug_overlay`
sh_type	SHT_ARM_DEBUGOVERLAY = SHT_LOPROC + 4 = 0x70000004
sh_flags	0
sh_addr	0
sh_offset	The section's file offset.
sh_size	The byte size of the section, a multiple of sh_entsize.
sh_link	0
sh_info	0
sh_addralign	0
sh_entsize	8 or 12 (the size of an entry).

The debug-overlay section is a table of fixed size rows, each row containing three values.

Table 2, The debug-overlay section row format

Field	Offset	Size	Value
dbg_offset	0	4	The offset in the debug section of the field containing the execution address.
dbg_shndx	4	**2** *4*	The index in the ELF file's section header table of a debug section that refers to an overlaid program section (via a potentially ambiguous execution address).
ov_shndx	6 8	**2** *4*	The index in the ELF file's section header table of the overlaid section referred to by the debug section.
sh_entsize		**8** *12*	If section indexes are smaller than SHN_XINDEX (`0xffff`). If any section index needs to be greater than SHN_XINDEX – 1.

Rationale

The size of many consolidated debug sections exceeds 2^{16} bytes so offsets need to be 4-byte quantities.

In reality, the indexes of consolidated sections will usually fit into 1 byte. However, a 6 byte entry does not fit well with the 4-byte alignment requirement of 4-byte offsets and saves little space compared with 8-byte entries.

A linker only needs to generate a section containing 12-byte entries when it would in any case need to generate a section of type SHT_SYMTAB_SHNDX in order to accommodate values of st_shndx greater than SHN_XINDEX – 1.

A linker should usually generate a debug-overlay section containing 8-byte entries.

3.3 Integration with GNU overlay management (speculative in r2.07)

The GNU debugger GDB features some support for debugging overlaid programs and defines a memory-resident table, identified by the _ovly_table symbol, for communicating between an overlay manager and GDB [GNU OV]. Each row in _ovly_table[] contains <*execution address, size, load address, loaded*> for an overlay segment.

From an embedded perspective there are a number of issues with this.

- The whole table must be writable (RAM) because the flag field *loaded* needs to be writable.
  In most embedded applications the other fields are read-only so they could reside in ROM.
- In a distributed overlay manager (e.g. each segment loads its successor explicitly) this data might need to replicated in _ovly_table[] just for the convenience of a debugger that could use a copy held on the host.
- It does not solve the problem of relating an overlaid program section to the debug sections that refer to it (for which --emit-relocs, a debug overlay section [§3.2.1], or similar, is needed).

To integrate this mechanism in a manner more useful to embedded systems we propose the following.

- Define a new `.ARM.overlay_table` section of type SHT_ARM_OVERLAYSECTION = 0x70000005 with contents exactly as defined by [GNU OV].
- The section header's sh_flags field contains SHF_ALLOC if the section resides in memory, otherwise the section is an offline section used by a debugger.
- If the sh_flags field contains SHF_ALLOC and *not* SHF_WRITE, the table resides in ROM.

  Otherwise the section resides in RAM and is used exactly as described by [GNU OV]. This is also the interpretation when the symbol _ovly_table exists but there is no `.ARM.overlay_table` section.

When the `.ARM.overlay_table` section exists and is not resident in RAM

- The *loaded* field of each _ovly_table entry is unused and the symbol _ovly_loaded identifies a separate *byte array* in RAM recording the *loaded* status of the corresponding overlay segments.