DWARF for the ARM® Architecture

Document number: ARM IHI 0040A, current through ABI release 2.06
Date of Issue: 5th May 2006, reissued 25th October 2007

Abstract

This document describes the use of the DWARF debug table format in the Application Binary Interface (ABI) for the ARM architecture.

Keywords

DWARF, DWARF 3.0, use of DWARF format

Latest release of this specification

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).

Licence

Proprietary notice

Contents

1 ABOUT THIS DOCUMENT

1.1 Change control

1.1.1 Current status and anticipated changes

This document has been released publicly. Anticipated changes to this document include:

- Typographical corrections.
- Clarifications.
- Compatible extensions.

1.1.2 Change history

Issue	Date	By	Change
1.0	30th October 2003	LS	First public release.
2.0	24th March 2005	LS	Second public release.
2.01	6th October 2005	LS	Added register numbers for VFP-v3 d0-d31 (§3.1).
2.02	5th May 2006	LS	Minor corrections now that DWARF 3.0 is a standard; incompatible changes to the values of DW_AT_endianity (§3.3) as a result.
A	25th October 2007	LS	Document renumbered (formerly GENC-003533 v2.02).

1.2 References

This document refers to, or is referred to by, the following documents.

Ref	External reference or URL	Title
AADWARF	*This document*	DWARF for the ARM Architecture.
BSABI		ABI for the ARM Architecture (Base Standard).
GDWARF	http://dwarf.freestandards.org	DWARF 3.0, the generic debug table format.

1.3 Terms and abbreviations

The *ABI for the ARM Architecture* uses the following terms and abbreviations.

Term	Meaning
AAPCS	Procedure Call Standard for the ARM Architecture
ABI	Application Binary Interface: 1. The specifications to which an executable must conform in order to execute in a specific execution environment. For example, the *Linux ABI for the ARM Architecture*. 2. A particular aspect of the specifications to which independently produced relocatable files must conform in order to be statically linkable and executable. For example, the *C++ ABI for the ARM Architecture*, the *Run-time ABI for the ARM Architecture*, the *C Library ABI for the ARM Architecture.*
AEABI	(Embedded) ABI for the ARM architecture (*this* ABI…)
ARM-based	… based on the ARM architecture …
core registers	The general purpose registers visible in the ARM architecture's programmer's model, typically r0-r12, SP, LR, PC, and CPSR.
EABI	An ABI suited to the needs of embedded, and deeply embedded (sometimes called *free standing*), applications.
Q-o-I	Quality of Implementation – a quality, behavior, functionality, or mechanism not required by this standard, but which might be provided by systems conforming to it. Q-o-I is often used to describe the tool-chain-specific means by which a standard requirement is met.
VFP	The ARM architecture's Vector Floating Point architecture and instruction set

1.4 Your licence to use this specification

Use, copying or disclosure by the US Government is subject to the restrictions set out in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of the Commercial Computer Software – Restricted Rights at 48 C.F.R. 52.227-19, as applicable.

This Specification is owned by ARM or its licensors and is protected by copyright laws and international copyright treaties as well as other intellectual property laws and treaties. The Specification is licensed not sold.

1. Subject to the provisions of Clauses 2 and 3, ARM hereby grants to LICENSEE, under any intellectual property that is (i) owned or freely licensable by ARM without payment to unaffiliated third parties and (ii) either embodied in the Specification or Necessary to copy or implement an applications binary interface compliant with this Specification, a perpetual, non-exclusive, non-transferable, fully paid, worldwide limited licence (without the right to sublicense) to use and copy this Specification solely for the purpose of developing, having developed, manufacturing, having manufactured, offering to sell, selling, supplying or otherwise distributing products which comply with the Specification.

2. THIS SPECIFICATION IS PROVIDED "AS IS" WITH NO WARRANTIES EXPRESS, IMPLIED OR STATUTORY, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF SATISFACTORY QUALITY, MERCHANTABILITY, NONINFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE. THE SPECIFICATION MAY INCLUDE ERRORS. ARM RESERVES THE RIGHT TO INCORPORATE MODIFICATIONS TO THE SPECIFICATION IN LATER REVISIONS OF IT, AND TO MAKE IMPROVEMENTS OR CHANGES IN THE SPECIFICATION OR THE PRODUCTS OR TECHNOLOGIES DESCRIBED THEREIN AT ANY TIME.

3. This Licence shall immediately terminate and shall be unavailable to LICENSEE if LICENSEE or any party affiliated to LICENSEE asserts any patents against ARM, ARM affiliates, third parties who have a valid licence from ARM for the Specification, or any customers or distributors of any of them based upon a claim that a LICENSEE (or LICENSEE affiliate) patent is Necessary to implement the Specification. In this Licence; (i) “affiliate” means any entity controlling, controlled by or under common control with a party (in fact or in law, via voting securities, management control or otherwise) and “affiliated” shall be construed accordingly; (ii) “assert” means to allege infringement in legal or administrative proceedings, or proceedings before any other competent trade, arbitral or international authority; (iii) “Necessary” means with respect to any claims of any patent, those claims which, without the appropriate permission of the patent owner, will be infringed when implementing the Specification because no alternative, commercially reasonable, non-infringing way of implementing the Specification is known; and (iv) English law and the jurisdiction of the English courts shall apply to all aspects of this Licence, its interpretation and enforcement. The total liability of ARM and any of its suppliers and licensors under or in relation to this Licence shall be limited to the greater of the amount actually paid by LICENSEE for the Specification or US$10.00. The limitations, exclusions and disclaimers in this Licence shall apply to the maximum extent allowed by applicable law.

ARM Contract reference LEC-ELA-00081 V2.0 AB/LS (9 March 2005)

1.5 Acknowledgements

This specification has been developed with the active support of the following organizations. In alphabetical order: ARM, CodeSourcery, Intel, Metrowerks, Montavista, Nexus Electronics, PalmSource, Symbian, Texas Instruments, and Wind River.

2 OVERVIEW

The ABI for the ARM architecture specifies the use of DWARF 3.0-format debugging data. For details of the base standard see [GDWARF].

The ABI for the ARM architecture gives additional rules for how DWARF 3.0 should be used, and how it is extended in ways specific to the ARM architecture. The following topics are covered in detail.

- The enumeration of DWARF register-numbers for, use in .debug_frame sections (§3.1).
- How the machine state (ARM state versus Thumb state) is encoded in DWARF 3.0 line number tables (§3.2).
- How to describe access to ARM architecture v6 *other-endian* data (§3.3).
- The definition of *Canonical Frame Address* (CFA) used by this ABI (§3.4).
- The generation and interpretation of debug frame Common Information Entries (§3.5).

2.1 Miscellaneous obligations on producers of relocatable files

2.1.1 Support for stack unwinding

To support stack unwinding by debuggers, producers must always generate .debug_frame sections, even when:

- Not generating other debug tables.
- At high optimization levels.
- Assembling hand-written assembly language, if that code calls code compiled from C or C++.

2.1.2 The debugging illusion (not mandatory)

Ideally, a user of a C/C++ source language debugger would like the illusion of:

- Stepping through the source program sequence point (SP) by sequence point.
- Being able to inspect the program's state at any sequence point, and seeing there the state predicted by the source language semantics.

For the purpose of debugging illusion, we define an *observation point* (OP) to be a point at which a debugger may (meaningfully) inspect a program's state. Most sequence points are also observation points. In addition

- There is an OP just after each function call (at the pc value to which the call will return).
- There is no OP at the SP after evaluation of function arguments but before the function call.

A variable's scope extends from the point of declaration of the identifier to the end of the smallest enclosing block. A variable need not have a value everywhere in its scope – it may be initialized some way after its declaration.

When a user signals to a producer (by Q-o-I means) that it should favour quality of debugging over quality of generated code, the producer should strive (Q-o-I) to generate DWARF tables and code supporting this illusion. Specifically:

- A statement should describe the code between consecutive OPs.
- At each OP, every in-scope, initialized, source code variable should have a location (need not be in memory), and that location should hold the value predicted by the source language semantics.

It is not necessary to describe OPs in code the producer knows can never be executed (e.g. in `if(0){i++;}`).

3 ARM-SPECIFIC DWARF DEFINITIONS

3.1 DWARF register names

[GDWARF] §2.6.1, *Register Name Operators*, suggests that the mapping from a DWARF register name to a target register number should be defined by the ABI for the target architecture. DWARF register names are encoded as unsigned LEB128 integers. Numbers 0-127 encode in 1 byte (grey rows below), 128-16383 in 2 bytes.

Table 1, Mapping from DWARF register number to ARM architecture register number

DWARF register number	ARM core or co-processor registers	Description
0–15	R0–R15	ARM core integer registers
16–63	None	**Obsolescent**: 16–47 were previously used for both FPA and VFP registers (Note 1)
64–95	S0–S31	**Legacy VFP-v2 use**: D0–D15 alias S0, S2, … S30 (Notes 1, 4)
96–103	F0–F7	**Obsolescent**: FPA registers 0-7 (Note 1)
104–111	wCGR0–wCGR7 ACC0–ACC7	Intel wireless MMX general purpose registers 0–7 XScale accumulator register 0–7 (Note 2)
112–127	wR0–wR15	Intel wireless MMX data registers 0–15
128	SPSR	Current SPSR register
129	SPSR_FIQ	FIQ-mode SPSR
130	SPSR_IRQ	IRQ-mode SPSR
131	SPSR_ABT	ABT-mode SPSR
132	SPSR_UND	UND-mode SPSR
133	SPSR_SVC	SVC-mode SPSR
134–143	None	Reserved for future allocation
144–150	R8_USR–R14_USR	User mode registers
151–157	R8_FIQ–R14_FIQ	Banked FIQ registers
158–159	R13_IRQ–R14_IRQ	Banked IRQ registers
160–161	R13_ABT–R14_ABT	Banked ABT registers
162–163	R13_UND–R14_UND	Banked UND registers
164–165	R13_SVC–R14_SVC	Banked SVC registers
166–191	None	Reserved for future allocation
192–199	wC0–wC7	Intel wireless MMX control register in co-processor 0–7
200–255	None	Reserved for future allocation
256-287	VFP-v3/Neon D0-D31	VFP-v3/Neon 64-bit register file (Note 4)
288-319	None	Reserved to VFP/Neon
320-8191	None	Reserved for future allocation
8192–16383	Vendor co-processor	Unspecified vendor co-processor register (Note 3)

Notes

1. In ADS toolkits, DWARF names 16–23 were used to represent FPA registers F0–F7 and 16-47 were used to represent VFP registers S0–S31. No application needs to use both numberings simultaneously but it can complicate decoding, so in RVDS new, non-overlapping, numbers 64-95 were allocated to VFP S0-S31. Debuggers that need to support legacy objects may need to handle both mappings.
2. Current implementations of the version 1 XScale Architecture specification implement only acc0, though eight such registers (acc0–acc7) are defined architecturally in co-processor 0. The version 2 specification defines the Wireless MMX co-processor in ARM co-processor slots 0 and 1. No system can contain both acc0 and MMX, so these numberings can overlap.
3. The vendor co-processor space is not specified by this ABI and should be used when there is unlikely to be a requirement for multiple vendors to support debugging such code. By using numbers in this space vendors can be sure that they will not conflict with future ABI allocations. If a set of co-processor registers is likely to be used directly from a high-level language and to require support of multiple toolkit vendors, then an application should be made to ARM for an allocation of a numbering in the reserved space.
4. The VFP-v3 and Neon architectures extend the register file to 32 64-bit registers, posing significant difficulties to extending the ABI v2.0 VFP encodings. There is no simple scheme using 1-byte register numbers that is compatible with the legacies. We have, therefore, introduced a new, simple, more precisely specified scheme using 2-byte register numbers. The new numbering scheme should also be used for VFP-v2.

The CPSR, VFP and FPA control registers are not allocated a numbering above. It is considered unlikely that these will be needed for producing a stack back-trace in a debugger.

3.1.1 VFP-v3 and Neon register descriptions

Architecturally, VFP-v3 and the Neon SIMD unit share a register file comprising 32 64-bit registers, D0-D31. Registers D0-D31 are described by DWARF register numbers 256-287. Register numbers 288-319 are reserved in case of future register file expansion.

DWARF registers 64-95 are *obsolescent* (and will become *obsolete* in the next major revision of the *ABI for the ARM Architecture*).

In DWARF terms:

- Register Dx is described as DW_OP_regx(256+x).
- Q registers Q0-Q15 are described by composing two D registers together.

  Qx = DW_OP_regx(256+2x) DW_OP_piece(8) DW_OP_regx(256+2x+1) [DW_OP_piece(8)]

  (Note that the final DW_OP_piece(8) can be omitted because the whole register is used. It is left in above for expositional clarity).
- S registers are described as bit-pieces of a register.
  - S[2x] = DW_OP_regx(256 + (x >> 1)) DW_OP_bit_piece(32, 0)
  - S[2x+1] = DW_OP_regx(256 + (x >> 1)) DW_OP_bit_piece(32, 32)
- Neon Half-word lanes and byte lanes are described in a similar way to S registers.

Producers should use this new numbering scheme for VFP-v2 before the ABI-v2.0 scheme (S0-S31 → 64-95) is declared obsolete. Consumers should accept both numberings for as long as there are legacy binaries.

3.2 DWARF line number information (ISA field)

[GDWARF] §6.2.5.2 *Standard Opcodes*, item 12, *DW_LNS_set_isa*, describes a single unsigned LEB128 operand that denotes the instruction set architecture (ISA) at the location identified by the line number table entry. The value of the operand is determined by the ABI for the architecture (this specification).

Under the ARM architecture there are many instruction set versions and variants, but few instruction set states. Under this ABI, the ISA field corresponding to a particular program address denotes the instruction set state encoded by the CPSR when the pc contains that address.

Table 2, DW_LNS_set_isa values for the ARM Architecture

Name	Value	Meaning
DW_ISA_UNKNOWN	0	I-set state not available or not recorded.
DW_ISA_ARM_thumb	1	T-bit will be set in the CPSR when pc contains this code address.
DW_ISA_ARM_arm	2	T-bit will be clear in the CPSR when pc contains this code address.
	Other	Reserved to the ABI for the ARM architecture.

3.3 Describing other endian data

ARM architecture version 6 allows programs to access data stored in the other byte order, either by executing REV* instructions, or by juggling the E bit in the PSR. Consequently, there is a need to describe in DWARF tables data that has been statically declared with a particular byte order.

This ABI mandates no particular way to describe the byte order of data manipulated by a programming language, but one could imagine a simple language extension like the following, or use of #pragma.

```
extern __big_endian T bx;     // bx contains big-endian data
extern __little_endian T lx;  // lx contains little-endian data
```

Usually, all data has the same byte order and this is recorded in the EI_DATA field of the header of the ELF file (as the value ELFDATA2MSB or ELFDATA2LSB).

To describe data that is explicitly declared big-endian or little-endian (by whatever means), use the DWARF 3.0 attribute `DW_AT_endianity` (0x65). It takes a single LEB128 constant argument value that is one of the following:

```
DW_END_default (= 0)
DW_END_big (= 1)          (Was 0 prior to the DWARF 3.0 standard)
DW_END_little (= 2)       (Was 1 prior to the DWARF 3.0 standard)
```

By default the ARM architecture is little endian, so `DW_END_default` should be interpreted as `DW_END_little`.

The `DW_AT_endianity` attributes can be attached to type entries as follows.

- Attached to a base type ([GDWARF], §5.1, *Base Type Entries*), this attribute gives the byte order of the data described by the base type.

  If this order differs from the default byte order recorded in the containing ELF file, a debugger should reverse the order of the bytes it fetches or stores when accessing values of that base type.

- Attached to any other type ([GDWARF], §5, *Type Entries*), this attribute indicates that the type was labeled explicitly (in some way) with the given byte order.

When representing such a type across its user interface, a debugger should label the representation in some way that indicates it was declared with an explicit byte order. Some possible labels for big-endian follow.

```
__big_endian T X;
__declspec(big_endian) T X;
T X __attribute__("big endian");

#pragma arm_big_endian
struct BigT { ... };
#pragma no_arm_big_endian
BigT X;
```

Any such representation by a debugger is entirely quality of implementation.

3.4 Canonical Frame Address

The term *Canonical Frame Address* (CFA) is defined in [GDWARF], §6.4, *Call Frame Information*.

This ABI adopts the typical definition of CFA given there.

- The CFA is the value of the stack pointer (r13) at the call site in the previous frame.

3.5 Common information entries

The DWARF virtual unwinding model is based, conceptually, on a tabular structure with one column for each target register ([GDWARF], §6.4.1, *Structure of Call Frame Information*). A .debug_frame Common Information Entry (CIE) specifies the initial values (on entry to an associated function) of each register.

The variability of processors conforming to the ARM architecture creates a problem for this model. A producer cannot reliably enumerate all the registers in the target. For example, an integer-only function might be included in one executable file for targets with VFP and another for targets without. In effect, it must be acceptable for a producer not to initialize, in a CIE, registers it does not know about. In turn this generates an obligation on consuming debuggers to default missing initial values.

This generates the following obligations on producers and consumers of CIEs.

Consumers must default the CIE initial value of any target register not mentioned explicitly in the CIE.

- Callee-saved registers (and registers unused by the program) should be initialized as if by DW_CFA_same_value, other registers as if by DW_CFA_undefined.

  A debugger can use built-in knowledge of the procedure call standard or can deduce which registers are callee-saved by scanning all CIEs.

To allow consumers to reliably default the initial values of missing entries by scanning a program's CIEs, without recourse to built-in knowledge, producers must identify registers not saved by callees, as follows.

- If a function uses *any* register from a particular hardware register class (e.g. ARM core registers), its associated CIE must initialize *all* the registers of that class that are not callee-saved to DW_CFA_undefined.

  (As an optimization, a producer need not initialize registers it can prove cannot be used by any associated functions and their descendants. Although these are not callee-saved, they are not callee-used either).

- If a function uses a callee-saved register R, its associated CIE must initialize R using one of the defined value methods (not DW_CFA_undefined).