Application Note 61

Big and Little Endian Byte Addressing

Document number: ARM DAI 0061A

Issued: March 1998

ENGLAND
Advanced RISC Machines Limited
Fulbourn Road
Cherry Hinton
Cambridge CB1 4JN
UK
Telephone: +44 1223 400400
Facsimile: +44 1223 400410
Email: info@arm.com

GERMANY
Advanced RISC Machines Limited
Otto-Hahn Str. 13b
85521 Ottobrunn-Riemerling
Munich
Germany
Telephone: +49 89 608 75545
Facsimile: +49 89 608 75599
Email: info@arm.com

JAPAN
Advanced RISC Machines K.K.
KSP West Bldg, 3F 300D, 3-2-1 Sakado
Takatsu-ku, Kawasaki-shi
Kanagawa
213 Japan
Telephone: +81 44 850 1301
Facsimile: +81 44 850 1308
Email: info@arm.com

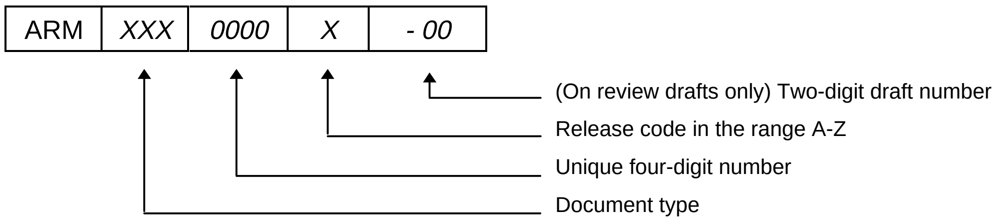
USA
ARM USA Incorporated
Suite 5
985 University Avenue
Los Gatos
CA 95030 USA
Telephone: +1 408 399 5199
Facsimile: +1 408 399 8854
Email: info@arm.com

World Wide Web address: http://www.arm.com

Key

Document Number

This document has a number which identifies it uniquely. The number is displayed on the front page and at the foot of each subsequent page.


ARM
XXX
0000
X
- 00
(On review drafts only) Two-digit draft number
Release code in the range A-Z
Unique four-digit number
Document type

Document Status

The document’s status is displayed in a banner at the bottom of each page. This describes the document’s confidentiality and its information status.

Confidentiality status is one of:

ARM Confidential	Distributable to ARM staff and NDA signatories only
Named Partner Confidential	Distributable to the above and to the staff of named partner companies only
Partner Confidential	Distributable within ARM and to staff of all partner companies
Open Access	No restriction on distribution

Information status is one of:

Advance	Information on a potential product
Preliminary	Current information on a product under development
Final	Complete information on a developed product

Change Log

Issue	Date	By	Change
A	March 1998	SKW	Released

Table of Contents

1 Introduction

The term *endianness* refers to the way in which multiple elements of identical size are stored in memory. There are two conventions for storing bytes within multi-byte quantities, such as 32-bit words, in byte-addressed memory. In a little-endian architecture, the least significant byte of the quantity is stored at the lowest memory address in the range of addresses used to store the quantity. The reverse is true in a big-endian architecture, where the most significant byte is stored at the lowest address.

The ARM can be configured to be either little- or big-endian. This Application Note discusses the effect that choosing either little- or big-endian configuration has on byte and halfword addressing.

For more information on endianness, please see:

- *Computer Architecture - A Quantitive Approach*, Henessey & Patterson, page 73, 2nd Ed, Publisher: Morgan Kaufman, ISBN: 1-55860-32908.
- *ARM System Architecture*, Steve Furber, page 110, 1st Ed, Publisher: Addison-Wesley, ISBN 0-201-40352-8.

Note *The basic endian configuration principles described in this application note are applicable to all ARM cores and AMBA-based systems. However, the signal names used are specific to ARM7TDMI. For equivalent signal names for other ARM cores, or AMBA-based systems, please refer to the appropriate datasheet.*

2 Configuring the Endianness of the ARM

The endian configuration of the ARM is controlled by the **BIGEND** signal to the core. This configures the byte and halfword extraction logic in the processor to extract the addressed data appropriately when accessing sub-word quantities. Tying this HIGH causes the ARM to operate in big-endian mode. Tying it LOW causes the ARM to operate in little-endian mode.

However, some ARM designs do not have the **BIGEND** signal accessible externally and instead the endianness of the ARM is controlled by the ARM MMU (coprocessor 15). For more information on how to control the endianness of such ARMs see Application Note 37*: Startup Configuration of ARM Processors with MMUs* (ARM DAI 0037).

Once you have configured your system's endianness, any application code built to run on that system must be built to match. To do this, use the compiler/assembler options `-li` (for a little-endian application—the default) or `-bi` (for a big-endian application) as appropriate. In addition, if you link with a standard ARM ANSI C library, this also needs to be of the same endianness.

Big-endian libraries have the extension `.32b` (ARM) or `.16b` (Thumb). Little-endian libraries have the extension `.32l` (ARM) or `.16l` (Thumb).

Note *The last character in the extension for a little-endian library is the letter* `l`*.*

When code is loaded into an ARM debugger, the debugger must also be instructed which endian configuration the system is in. With armsd this is done using the `-li` (default) or `-bi` option as appropriate. In the ARM Debugger for Windows, it is done in the **Options->Configure Debugger** dialog box.

Note *The ARM Software Development toolkit does not support dynamic endian configuration. The whole application code must be built either as big- or as little-endian. An application cannot be built as a combination of the two, nor can it change the endianness of the system during normal execution.*

3 Effect of Endian Configuration on the System

3.1 Connection to memory

The endian configuration of your system determines how you should connect the memory up to the ARM. If you do not connect memory up to match the configuration of **BIGEND**, byte and halfword accesses (including Thumb instructions fetches) will not behave as predicted.

For narrow memory subsystems (that is, 8 or 16 bits wide) the bytes must always be presented to the correct byte lanes on the 32-bit microprocessor core.

Also note that separate byte write enables must be provided to the memory (you should not utilize 32-bit wide, on-chip memory with only a single write-enable control for the whole word).

See ***4.1 Connection to memory*** on page 7 for details of how to connect memory in a little-endian system. See ***5.1 Connection to memory*** on page 9 for details of how to connect memory in a big-endian system.

3.2 Word accesses to memory

Unaligned word stores are effectively truncated to word-aligned addresses, such that the bottom two bits of the address are equal to zero. Thus word stores are always made to word-aligned addresses. This means that bit 31 of the register being stored always appears on D[31].

With aligned word loads, D[31] is always loaded into bit 31 of the register being loaded. However, word loads from unaligned addresses load data from memory in a way that depends upon the endian configuration of the system and the offset from the truncated word-aligned address.

The following code can be used to see the effect of unaligned word loads:

```
    AREA unalign, CODE, READONLY
    ENTRY
    MOV r2, #0xF000
    LDR r3, =0x7654321
    STR r3, [r2]      ; store the word
    LDR r4, [r2]      ; read word back - aligned
    LDR r5, [r2,#1]   ; read word - unaligned - offset 1
    LDR r6, [r2,#2]   ; read word - unaligned - offset 2
    LDR r7, [r2,#3]   ; read word - unaligned - offset 3
exit
    MOV  r0, #0x18    ; angel_SWIreason_ReportException
    LDR  r1, =0x20026 ; ADP_Stopped_ApplicationExit
    SWI  0x123456     ; Angel semihosting ARM SWI
    END
```

Note that the ARM compilers sometimes make use of unaligned loads. For instance, an unaligned load can be used to load a short into the top 16 bits of a register. This can then be shifted down into the bottom 16 bits using an LSR operation which automatically clears the top 16 bits to zero (as required). This could occur in situations where there is no direct halfword support on a core, or where the halfword is in a packed struct and occupies the middle two bytes of the word.

It is possible to disable this feature using the compiler option `-za1`.

If you do not want to allow unaligned word accesses, the memory controller would be normally be responsible for generating the fault (typically a data abort). However this is not usually a sensible thing to do. This is because when the ARM is fetching instructions, A[1:0] (for ARM code) or A[0] (for Thumb code) may hold any value, so it is quite likely that instruction fetches may appear not to be correctly aligned. Therefore if this type of fault is generated, it must only be generated during data transfers.

This adds complexity to the memory controller(s), as they now need to decode `A[1:0]`, `MAS[1:0]` (or `nBW` on cores which do not support halfword load/store instructions), and `nOPC`, to generate an abort. If you want to do this on loads only, you must use `nRW` as well (although if you generate such faults you should strictly fault reads and writes).

On ARM designs with an ARM MMU (coprocessor 15) the faulting of misaligned word accesses can also be caused by the MMU. This feature can be turned on by MMU coproc control register, bit ‘1’ (‘A’).

See ***4.2 Word accesses to memory*** on page 7 for details of word accesses to memory in a little-endian system. See ***5.2 Word accesses to memory*** on page 9 for details of word accesses to memory in a big-endian system.

3.3 Byte and halfword accesses to memory

When the ARM stores a byte quantity to memory, it outputs the same byte across all four byte lanes. The memory controller must therefore decode `A[1:0]` appropriately and latch it into the correct byte lane of your memory system. This also applies with halfword stores, with the halfword being output across both halves of the data bus.

For details of the required decoding see ***4.1 Connection to memory*** on page 7 for little-endian systems and ***5.1 Connection to memory*** on page 9 for big-endian systems.

For byte reads, the ARM expects to read the data from byte lane 0 for a word-aligned address, byte lane 1 for an offset of 1 and so on. For halfword reads, the ARM expects to read the data from byte lanes 0 and 1 for a word-aligned address, and byte lanes 2 and 3 for a read offset by 2.

Note *The affect of unaligned halfword load and store instructions is unpredictable. The results of such instructions should not be relied upon and they should therefore be avoided.*

If the ARM core being used is Thumb-compatible, the way that Thumb instructions will be fetched from memory is also controlled by the endianness of the system. The addressed halfwords are always accessed in sequential low-halfword, high-halfword order.

The following code can be used to see the effect of byte and halfword accesses:

```
    AREA subword, CODE, READONLY
    ENTRY
    MOV r2, #0xF000     ; address to store word
    LDR r3, =0x76543210 ; value to store
    STR r3, [r2]        ; store the word
    LDR r4, [r2]        ; read word back
    LDRB r5, [r2]       ; read byte 0
    LDRB r6, [r2,#1]    ; read byte 1
    LDRB r7, [r2,#2]    ; read byte 2
    LDRB r8, [r2,#3]    ; read byte 3
    LDRH r9, [r2,#0]    ; read halfword
    LDRH r10,[r2,#2]    ; read halfword
exit
    MOV  r0, #0x18    ; angel_SWIreason_ReportException
    LDR  r1, =0x20026 ; ADP_Stopped_ApplicationExit
    SWI  0x123456     ; Angel semihosting ARM SWI
    END
```

4 Little-Endian Operation

4.1 Connection to memory

A little-endian configured ARM should be connected to memory as follows:

- Byte 0 of the memory connected to D[7:0]
- Byte 1 of the memory connected to D[15:8]
- Byte 2 of the memory connected to D[23:16]
- Byte 3 of the memory connected to D[31:24]

The byte write enables are decoded:

nRW	MAS[1:0]	A[1:0]	we31to24	we23to16	we15to8	we7to0
0	? ?	? ?	0	0	0	0
1	0 0	0 0	0	0	0	1
1	0 0	0 1	0	0	1	0
1	0 0	1 0	0	1	0	0
1	0 0	1 1	1	0	0	0
1	0 1	0 ?	0	0	1	1
1	0 1	1 ?	1	1	0	0
1	1 0	? ?	1	1	1	1
1	1 1	? ?	--------------------------- Reserved ---------------------------			

Note *During halfword and byte transfers, the address bits marked '?' should be ignored.*

4.2 Word accesses to memory

In a little-endian system a LDR from a non-word-aligned address causes the data to be rotated into the register so that the addressed byte occupies bits 0–7 of the register.

Pictorially, if you have an aligned word in memory:

Databus	31–24	23–16	15–8	7–0
Byte	3	2	1	0
Value	76	54	32	10

Loading this in at each possible offset gives:

Load offset	Register contents
0x0	0x76543210
0x1	0x10765432
0x2	0x32107654
0x3	0x54321076

The situation with STRs is simpler, as bit 31 of the register being stored always appears on databus output 31.

This can be seen by building and running the program given in ***3.2 Word accesses to memory*** on page 4. Upon program completion, registers r4 to r7 can be seen to contain the values given above.

```
    armasm -li word.s
    armlink word.o -o word
    armsd -li word

A.R.M. Source-level Debugger vsn 4.48 (Advanced RISC Machines SDT 2.11)
[Sep  9 1997]
ARMulator 2.02 [Sep  9 1997]
ARM7TDM, 4GB, Dummy MMU, Soft Angel 1.4 [Angel SWIs, Demon SWIs], FPE,
  Profiler, Tracer, Pagetables, Little endian.
Object program file word
ARMSD: go
Program terminated normally at PC = 0x000080a4 (unalign + 0x24)
+0024 0x000080a4: 0xef123456  V4.. :    swi      0x123456
ARMSD: reg
  r0  = 0x00000018  r1  = 0x00020026  r2  = 0x0000f000  r3  = 0x76543210
  r4  = 0x76543210  r5  = 0x10765432  r6  = 0x32107654  r7  = 0x54321076
  r8  = 0x00000000  r9  = 0x00000000  r10 = 0x00000000  r11 = 0x00000000
  r12 = 0x000080b0  r13 = 0x00000000  r14 = 0x00008010
  pc  = 0x000080a4  psr = %nZCvift_User32
```

4.3 Byte and halfword accesses to memory

When fetching Thumb instructions, the addressed halfwords are always accessed in sequential low-halfword, high-halfword order. In little-endian configuration this is D[15:0] then D[31:16].

The effect of reading bytes and halfwords from various offsets in memory can be seen by building and running the program given in ***3.3 Byte and halfword accesses to memory*** on page 5.

```
    armasm -li -arch 4 subword.s
    armlink subword.o -o subword
    armsd -li subword

A.R.M. Source-level Debugger vsn 4.48 (Advanced RISC Machines SDT 2.11)
[Sep  9 1997]
ARMulator 2.02 [Sep  9 1997]
ARM7TDM, 4GB, Dummy MMU, Soft Angel 1.4 [Angel SWIs, Demon SWIs], FPE,
  Profiler, Tracer, Pagetables, Little endian.
Object program file subword
ARMSD: go
Program terminated normally at PC = 0x000080b0 (subword + 0x30)
+0030 0x000080b0: 0xef123456  V4.. :    swi      0x123456
ARMSD: reg
  r0  = 0x00000018  r1  = 0x00020026  r2  = 0x0000f000  r3  = 0x76543210
  r4  = 0x76543210  r5  = 0x00000010  r6  = 0x00000032  r7  = 0x00000054
  r8  = 0x00000076  r9  = 0x00003210  r10 = 0x00007654  r11 = 0x00000000
  r12 = 0x000080bc  r13 = 0x00000000  r14 = 0x00008010
  pc  = 0x000080b0  psr = %nZCvift_User32
```

5 Big-Endian Operation

5.1 Connection to memory

A big-endian configured ARM should be connected to memory as follows:

- Byte 0 of the memory connected to D[31:24]
- Byte 1 of the memory connected to D[23:16]
- Byte 2 of the memory connected to D[15:8]
- Byte 3 of the memory connected to D[7:0]

The byte write enables are decoded:

nRW	MAS[1:0]	A[1:0]	we31to24	we23to16	we15to8	we7to0
0	? ?	? ?	0	0	0	0
1	0 0	0 0	1	0	0	0
1	0 0	0 1	0	1	0	0
1	0 0	1 0	0	0	1	0
1	0 0	1 1	0	0	0	1
1	0 1	0 ?	1	1	0	0
1	0 1	1 ?	0	0	1	1
1	1 0	? ?	1	1	1	1
1	1 1	? ?	--------------------------- Reserved ---------------------------			

Note *During halfword and byte transfers, the address bits marked '?' should be ignored.*

5.2 Word accesses to memory

In a big-endian system a LDR from a non-word-aligned address causes the data to be rotated into the register so that the addressed byte occupies:

- bits 31–24 of the register with an offset of 2 (or 0).
- bits 15–8 of the register with an offset of 1 or 3.

Pictorially, if you have an aligned word in memory:

Databus	31–24	23–16	15–8	7–0
Byte	0	1	2	3
Value	76	54	32	10

Loading this in at each possible offset gives:

Load offset	Register contents
0x0	0x76543210
0x1	0x10765432
0x2	0x32107654
0x3	0x54321076

The situation with STRs is simpler, as bit 31 of the register being stored will always appear on databus output 31.

This can be seen by building and running the program given in ***3.2 Word accesses to memory*** on page 4. Upon program completion, registers r4 to r7 can be seen to contain the values given above.

```
    armasm -bi word.s
    armlink word.o -o word
    armsd -bi word

A.R.M. Source-level Debugger vsn 4.48 (Advanced RISC Machines SDT 2.11)
[Sep  9 1997]
ARMulator 2.02 [Sep  9 1997]
ARM7TDM, 4GB, Dummy MMU, Soft Angel 1.4 [Angel SWIs, Demon SWIs], FPE,
  Profiler, Tracer, Pagetables, Big endian.
Object program file word
ARMSD: go
Program terminated normally at PC = 0x000080a4 (unalign + 0x24)
+0024 0x000080a4: 0xef123456  ..4V :    swi      0x123456
ARMSD: reg
  r0  = 0x00000018  r1  = 0x00020026  r2  = 0x0000f000  r3  = 0x76543210
  r4  = 0x76543210  r5  = 0x10765432  r6  = 0x32107654  r7  = 0x54321076
  r8  = 0x00000000  r9  = 0x00000000  r10 = 0x00000000  r11 = 0x00000000
  r12 = 0x000080b0  r13 = 0x00000000  r14 = 0x00008010
  pc  = 0x000080a4  psr = %nZCvift_User32
```

5.3 Byte and halfword accesses to memory.

When fetching Thumb instructions, the addressed halfwords are always accessed in sequential low-halfword, high-halfword order. In big-endian configuration this is D[31:16], then D[15:0].

The effect of reading bytes and halfwords from various offsets in memory can be seen by building and running the program given in ***3.3 Byte and halfword accesses to memory*** on page 5.

```
    armasm -bi -arch 4 subword.s
    armlink subword.o -o subword
    armsd -bi subword

A.R.M. Source-level Debugger vsn 4.48 (Advanced RISC Machines SDT 2.11)
[Sep  9 1997]
ARMulator 2.02 [Sep  9 1997]
ARM7TDM, 4GB, Dummy MMU, Soft Angel 1.4 [Angel SWIs, Demon SWIs], FPE,
  Profiler, Tracer, Pagetables, Big endian.
Object program file subword
ARMSD: go
Program terminated normally at PC = 0x000080b0 (subword + 0x30)
+0030 0x000080b0: 0xef123456  ..4V :    swi      0x123456
ARMSD: reg
  r0  = 0x00000018  r1  = 0x00020026  r2  = 0x0000f000  r3  = 0x76543210
  r4  = 0x76543210  r5  = 0x00000076  r6  = 0x00000054  r7  = 0x00000032
  r8  = 0x00000010  r9  = 0x00007654  r10 = 0x00003210  r11 = 0x00000000
  r12 = 0x000080bc  r13 = 0x00000000  r14 = 0x00008010
  pc  = 0x000080b0  psr = %nZCvift_User32
```