ARM1176JZ-S™ and ARM1176JZF-S™

Programmer Advice Notice

Use of BLX (immediate)

ARM reference 760522

ARM®

ARM1176JZ-S and ARM1176JZF-S

Programmer Advice Notice

Use of BLX (immediate)

ARM reference 760522

Release Information

The following changes have been made to this document.

Change history

Date	Issue	Confidentiality	Change
12 July 2011	A	Non-Confidential	First release

Proprietary notice

Confidentiality status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Product Status

The information in this document is final, that is for a developed product.

Feedback on content

If you have any comments on content, then send an e-mail to errata@arm.com. Give:

- the title
- the number
- the page numbers to which your comments apply
- a concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Web Address

`http://www.arm.com`

Table of Contents

1 Preface

1.1 Intended audience

This document is intended for operating system developers, application developers and static toolchain developers who need to implement software workarounds for a possible issue, identified as ARM reference 760522, that can apply to execution of a Thumb BLX (immediate) instruction on an ARM1176JZ-S or ARM1176JZF-S processor.

1.2 Document status

This is a released document. However it might include technical inaccuracies or typographical errors. See *Feedback on content* on page 2.

1.3 References

Document number	Title
ARM IHI 0044	*ELF for the ARM Architecture*
ARM DUI 0493	*ARM® Compiler toolchain Linker Reference*

2 Terms and abbreviations

This document uses the following terms and abbreviations.

Term	Description
Static toolchain	Toochain consisting of at least a compiler or static linker. For example RVCT is a static toolchain with armcc as the compiler, armlink as the linker.
Veneer	A series of instructions inserted between a branch target and its destination. These are often used to change state and or extend the range of branch instructions.
Object Producer	A tool that generates an ELF Object. A compiler is an Object Producer.
Section	An ELF section is the smallest unit of data on which a linker operates. An Object Producer must make any inter-section references explicit by relocation directives. Intra-section references are not required to be exposed to the linker.
Relocation Directive	An instruction from the Object Producer to the linker to fix up code or data when final addresses are known.
ABI Compliant ELF	An Object that conforms to *ELF for the ARM Architecture*.

3 Introduction

This Programmer Advice Notice describes a problem that might occur when using the BLX (immediate) instruction on the ARM1176JZ-S or ARM1176JZF-S processors. It describes software and static toolchain workarounds for the problem.

4 Problem description

4.1 Processors affected

All revisions of the ARM1176JZ-S and ARM1176JZF-S processors are affected.

This problem cannot affect any other ARM11 processor. This means that the following processors are not affected:

- ARM1136
- ARM1156
- ARM11 MPCore.

4.2 Instruction sequences affected

The affected sequence is a BLX (immediate) instruction from Thumb to ARM to which all of the following apply:

- the target ARM instruction is aligned on a doubleword boundary
- the target ARM instruction is not an unconditional branch
- the target ARM instruction is not an unconditional load to the PC
- the BLX instruction is aligned at 30 or 0 modulo 32, or is the target of a branch.

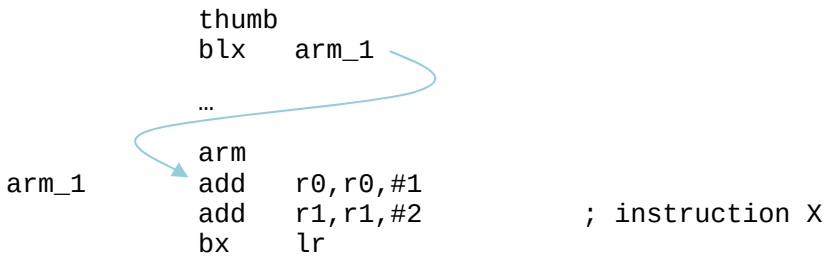
In very rare circumstances that depend on the validity of entries in the instruction cache, the validity of entries in the branch target address cache (BTAC), and code alignments, an instruction sequence that meets the above conditions can execute erroneously. Therefore, software must avoid using these sequences.

Note: The problem does not exist for ARM to Thumb transitions when using BLX (immediate).

4.3 Example code

4.3.1 Example of unsafe code

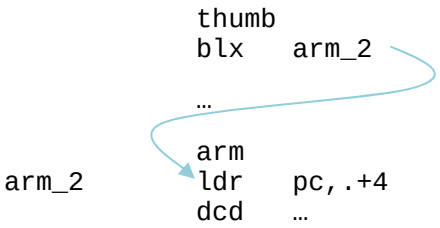
The following code might be unsafe:



Under certain conditions, an incorrect bitpattern is fetched for `instruction X`.

4.3.2 Example of safe code

The following code is safe:



This sequence is safe because the destination of the BLX instruction branches unconditionally.

Note: Some dynamic program linkage sequences are of this form.

5 OS Developers

Operating System developers that are affected by this problem must apply one of the application code workarounds described in section 6 to any affected assembler code, and then recompile any affected C code using an updated toolchain. A suitable updated toolchain is the ARM Compiler toolchain version 4.1p5 or later, using the `--no_blx_thumb_arm` switch.

Static Toolchain Developers on page 11 describes the requirements for a third-party toolchain workaround for this issue.

5.1 Dynamic linking

Many platforms support a procedure for calling functions in another link unit such as a shared object or DLL. Often, these calls are indirected using a *Procedure Linkage Table* (PLT). The PLT is a table of veneers that ensure that each call:

- finds the correct function
- is in range
- is in the correct state.

In software for an ARM1176 processor, PLT veneers are implemented in ARM state as they require access to the high registers, R8-R12. This means a call from Thumb state using the PLT must change state. This might involve a `BLX` instruction.

Applying the workarounds prevents the generation of the `BLX`. However, this can result in a veneer that is between the Thumb call and the PLT, resulting in performance loss. With some small changes, PLT sections can be generated such that veneers are not required.

5.1.1 ARM Linux style PLT sequences

An ARM Linux PLT consists of a special first entry to handle lazy loading using a call to the Dynamic Loader. Control is passed to this entry only by another PLT entry that requires lazy loading. The other PLT entries are potential ARM `BLX` targets from Thumb code.

The GCC implementation of these sequences is:

PLT entry	PLT sequence

0	`Entry 0`
	`$a`
	`PUSH {lr}`
	`LDR lr,[pc,#4]`
	`ADD lr,pc,lr`
	`LDR pc,[lr,#8]!`
	`$d`
	`DCD <GOT address>`

1 .. N	`$a`
	`ADD ip, pc, #0`
	`ADD ip, ip, #<offset1>`
	`LDR pc, [ip, #<offset2>]!`

In this table, the size of the first PLT entry is 5 words, and all other entries are 3 words. Therefore, alternate PLT entries are doubleword aligned.

The following simple modifications to the PLT entries ensure that the `BLX` targets are non-doubleword aligned:

1. Increase the section alignment of the `.plt` section to doubleword alignment.
2. Increase the size of the entries, other than entry 0, to four words, for example by adding a word of padding after the `LDR pc`.

5.1.2 Symbian style PLT entries

The Symbian PLT entries are a table of indirect branches. For example:

```
LDR pc, [pc, #-4]
DCD destination
```

The BLX target is itself an indirect branch, therefore the problem cannot occur. No change is required to the PLT sequence.

5.1.3 Other possible PLT entries

Many platforms have a sequence similar to the one for Symbian but with the destination address in read/write data. This requires an additional load.

```
.plt
        LDR ip, [pc, #0]
        LDR pc, [ip, #0]
        DCD <address of GOT entry for function>
        ...
.got
        DCD <address of function>
```

The BLX target is not a branch so the problem might occur. The entry size for the PLT is 3 words. This means that alternate words in the table are doubleword aligned.

The following simple modifications to the PLT entries ensure that the BLX targets are non-doubleword aligned:

1. Increase the section alignment of the `.plt` section to doubleword alignment.
2. Increase the size of the entries, other than entry 0, to four words, for example by adding a word of padding before the `LDR ip`.

6 Application Developers

Application developers that are affected by this problem must apply one of the following workarounds to any affected assembler code, and recompile any affected C code using an updated toolchain. A suitable updated toolchain is the ARM Compiler toolchain version 4.1p5 or later, using the `--no_blx_thumb_arm` switch.

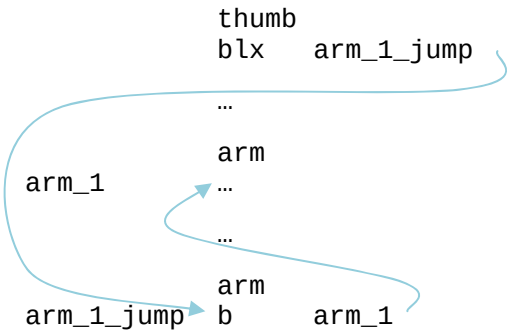
Static Toolchain Developers on page 11 describes the requirements for a third-party toolchain workaround for this issue.

6.1 Workarounds

Workarounds for the problem involve modifying the code to remove one or more of the preconditions for the problem. This can be done with minimal impact on application performance.

6.1.1 BLX to an ARM branch instruction

Change the BLX (immediate) to transfer to an ARM unconditional branch instruction that branches to the target ARM routine:



You can place the ARM branch instruction anywhere within range of the Thumb BLX instruction (± 16Mb) and the target ARM routine (± 32Mb).

An instance of this workaround has:

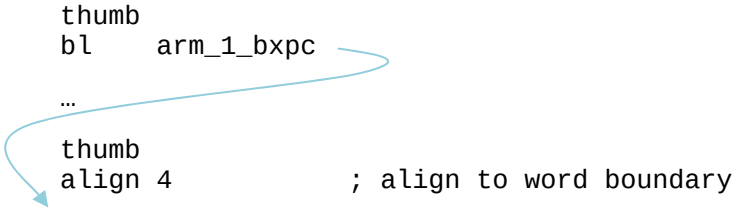
- a code-size penalty of 4 bytes for each ARM routine called
- an execution-time penalty of 1 or 4 cycles per call, depending on whether the ARM branch is predicted.

6.1.2 BL to a Thumb-to-ARM state-change instruction (BX PC)

Change the BLX (immediate) into a BL (immediate) to a state-change instruction, that is, Thumb BX PC followed by 2-byte padding. This is an alternative way to achieve the state-change required when calling an ARM routine from Thumb code.

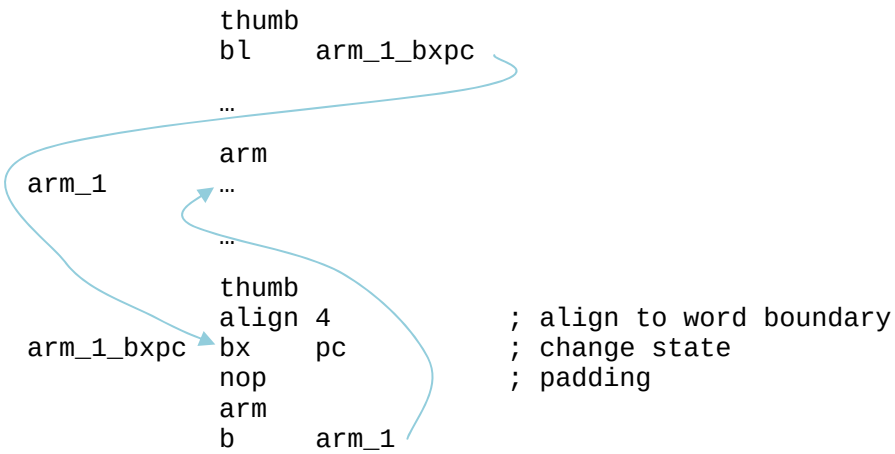
Note: The Thumb BX PC instruction must be word aligned.

If it is possible to recompile or reassemble, and there is no fallthrough from the preceding code, place the state-change sequence immediately before the target ARM routine:



```
arm_1_bxpc  bx    pc          ; change state
            nop               ; padding
            arm
arm_1       …                 ; original entry point
```

Otherwise, place the BX PC instruction at a different location, and follow it with a branch to the target ARM routine:



If the Thumb BX PC can be placed immediately before the ARM routine, this workaround has a code-size penalty of 4 bytes and an execution-time penalty of 5 cycles.

If the Thumb BX PC must be followed by an ARM branch to the ARM routine, this workaround has a code-size penalty of 8 bytes and an execution-time penalty of 6 or 9 cycles, depending on whether the ARM branch is predicted.

6.1.3 Align the ARM routine on an odd word

It might be possible to align the target routine on an odd word boundary. This generally requires recompilation or reassembly of the source.

Using the ARM assembler:

```
            arm
            align 8,4
arm_1       …
```

If the routine is entered by fallthrough, make sure you define the section with the CODEALIGN option introduced in RVCT 3.1, so that alignment padding uses NOP instructions.

7 Static Toolchain Developers

7.1 ARM Compiler Toolchain support

For details of workaround support in the ARM Compiler Toolchain, see the latest ARM Compiler Toolchain documentation on the ARM website. The `--no_blx_thumb_arm` switch provides this support, and is described in the *ARM® Compiler toolchain Linker Reference*. This switch is supported in version 4.1p5 or later, of the ARM Compiler Toolchain. For earlier versions, contact ARM support.

7.2 Detecting the problem in a static toolchain

The conditions for the ARM1176JZ-S and ARM1176JZF-S problem that can be detected by a static toolchain are as follows. The problem can occur only when all of these conditions are met:

Condition	Description
3	The BLX (immediate) must be executed in Thumb state to switch to ARM state
4	The BLX (immediate) must be either: a. the target of a branch instruction b. located at an address whose value modulo 32 is equal to 30 or 0
5	The target of the BLX (immediate) is a doubleword aligned address
6	The target of the BLX (immediate) is not an unconditional branch

Note: Conditions 1 and 2 are ARM internal designations that are not relevant to this document.

7.3 Use of BLX instruction by toolchains

Software for execution on an ARM1176JZ-S or ARM1176JZF-S processor can contain only ARM instructions, or a mixture of ARM and Thumb instructions.

To support mixed ARM and Thumb instructions, a toolchain must support state changes between ARM and Thumb state. The ARM architecture provides a number of instructions that change state, one of which is the BLX (immediate) instruction.

To safely use a BLX (immediate) instruction a toolchain must know:

1. The state of the caller.
2. The state of the callee.
3. The distance between the caller and the callee.

An object producer such as a compiler has information about:

1. The state of the caller.
2. The state of the callee, if it resides in the same source file.
3. The distance between the caller and the callee if the callee resides in the same ELF section of the object file.

It is rare for a compiler to change state within a single object file. Instead of generating a BLX instruction, the compiler generates a BL instruction with an appropriate relocation directive. A linker can use the relocation directive to change a BL to a BLX if conditions permit this change.

This document assumes that a compiler encodes all function calls with a relocation directive. The linker can then choose between the BL or BLX instruction. This assumption means a linker can fix the problem without disassembling the image.

For toolchains that do not generate state change veneers at link time, see *Procedure for generating state change veneers* on page *16*.

7.3.1 Hidden BLX instructions

An assembler can generate a BLX instruction to change between ARM and Thumb instructions within a single section. If this happens, whether a relocation directive makes the BLX instruction visible to the linker is dependent on the assembler.

A compiler can produce a BLX instruction without a relocation directive when there is a state change within an ELF section. However, ARM does not know any compilers that do this.

This document assumes that it is possible to hand-check assembly language for Thumb to ARM BLX instructions that do not have a relocation directive.

A toolchain that generates many unrelocated BLX instructions might require the additional step of disassembling the code to convert unrelocated BLX instructions to relocated BLX instructions.

7.4 Workarounds

Workarounds for the problem involve modifying the code to remove one or more of the preconditions for the problem. Toolchain developers can choose from the workarounds described in this section. Each workaround has minimal impact on code size and application performance.

7.4.1 Ensure Thumb BLX (immediate) targets are not doubleword aligned.

The problem can occur only if the target of a Thumb BLX (immediate) instruction is doubleword aligned. Therefore, the problem can be fixed by ensuring that the target of any Thumb BLX (immediate) instruction is not doubleword aligned.

ARM instructions must be at least word aligned. So in most programs alternate ARM functions are doubleword aligned. There are a number of steps that a toolchain can make to prevent doubleword alignment.

7.4.1.1 Compiler

The compiler places compiled code in sections. These sections have a required alignment that the linker must follow when giving the section its final address. To ensure that an ARM function is not doubleword aligned, the compiler must increase the alignment of each section to at least doubleword alignment, and place all ARM code symbols so that they are not doubleword aligned.

It is not always possible to apply this transformation. The programmer can request doubleword or higher alignment for the address of the ARM function that the toolchain must follow.

7.4.1.2 Assembler

There is no automatic fix that the assembler can make. You must manually apply the same translation as the compiler.

7.4.1.3 Linker

A linker can automatically convert word-aligned sections to doubleword alignment. With the addition of one word of padding at the start of the section, the linker can change the offset of each instruction in the section. This means a linker can always transform a word-aligned

section that contains only one ARM BLX target such that the ARM code symbol is not doubleword aligned.

A linker can only modify the start position of a section in multiples of the original section alignment. Therefore if the section containing the ARM code is already doubleword aligned, the linker cannot change the doubleword alignment of any instructions within the section.

7.4.1.4 Libraries and third-party objects

A linker cannot guarantee the non-doubleword alignment of ARM BLX targets. Any binary-only library code shipped with the toolchain must not have any instances of a doubleworld aligned ARM BLX target.

There is no guaranteed solution for third-party objects that cannot be recompiled.

7.4.1.5 Recommendation

To use alignment changes alone, modifications are required in the compiler, linker, and libraries. The workaround cannot fix third-party objects that cannot be recompiled. This document recommends that you use this workaround only if the toolchain does not handle state changes at link time.

7.4.2 Use ARM v4T interworking to change from Thumb to ARM state

The problem can be fixed by ensuring that Thumb BLX (immediate) instruction is not used.

A BLX (immediate) instruction cannot handle all the types of state transition that *ELF for the ARM Architecture* requires. For example:

- the distance from source to target might exceed the range of the BLX instruction
- the source instruction might be a B instruction that cannot be transformed to a BLX because of link register corruption.

Most toolchains already have support for Thumb to ARM transitions that do not involve a BLX instruction.

The simplest possible change to a toolchain is to use one of the alternative interworking transitions from Thumb to ARM. The smallest and fastest veneers that a toolchain can generate are the Thumb to ARM inline veneer and the Thumb to ARM short veneer.

The following table shows example code for transforming all relocated BLX instructions to BL instructions.

Original	**Replacement**

`$t ; Thumb state`
`  BLX arm_target`
`…`
`arm_target: ; Must be at offset 0`
`  ; from a Section start`
	`$t ; Thumb state`
	`  BL $Veneer$TA$I$$arm_target`
	`  …`
	`$Veneer$TA$I$$arm_target`
	`  BX PC`
	`  NOP`
	`; Fall through`
	`arm_target`

`$t ; Thumb state`
`  BLX arm_target`
`…`
`arm_target: ; Can be at any offset`
`  ; from a Section start`
`…`
	`$t ; Thumb state`
	`  BL $Veneer$TA$S$$arm_target`
	`  …`
	`$ Veneer$TA$S$$arm_target:`
	`$t ; Thumb state`
	`  BX PC`
	`  NOP`
	`$a ; ARM state`
	`  B arm_target`

Transforming all relocated BLX instructions to BL instructions avoids the preconditions for the problem. To minimize risk, this document recommends that modifications to toolchains that are not in active development use this fix.

7.4.2.1 Impact of using ARM v4T interworking for Thumb to ARM state changes

Each BLX from a Thumb instruction to an ARM instruction must go through either an inline or a short branch veneer. The performance impact of each varies. For example, an extra instruction cache miss might occur because the veneer is not close to the target.

All Thumb callers in range of the veneer can share a veneer for an ARM target. In ideal conditions, a linker can place the veneer within range of all of the original BLX instructions.

The following table shows the extra instructions required, the performance impact in cycles, and the code size impact in bytes for an inline and short branch veneer.

Veneer	Extra instructions executed	Performance impact in cycles	Code size impact in bytes
Inline	BX PC	5-7	4
Short branch	BX PC B	5-7 1-4 depending on whether the branch is predicted	8

7.4.3 Use a custom problem-avoiding veneer

Condition 6 for the problem to occur, given in *Detecting the problem in a static toolchain*, is that the target of the BLX instruction is not an unconditional branch instruction. A toolchain can use this condition to create an alternative state change veneer that makes use of BLX safely.

The following table shows the original and replacement code.

Original	Replacement

`$t ; Thumb state`
`  BLX arm_target`
`  …`
`arm_target: ; Can be at any offset`
`  ; from a Section start`
`  …`
	`$t ; Thumb state`
	`  BLX $Veneer$AA$S$$arm_target`
	`…`
	`$Veneer$AA$S$$arm_target`
	`  B arm_target`
	`…`
	`arm_target`

7.4.3.1 Impact of using custom problem-avoiding veneer

The following table shows the extra instructions required, the performance impact in cycles, and the code size impact in bytes for a custom veneer.

Veneer	Extra instructions executed	Performance impact in cycles	Code size impact in bytes
Custom	B	1-4 depending on whether the branch is predicted	4

The custom veneer is more efficient to use than the v4T interworking because it avoids BX PC.

7.4.4 Use a combination of alignment and veneer changes

Condition 5 for the problem to occur, given in *Detecting the problem in a static toolchain*, is that the target of the BLX (immediate) instruction must be doubleword aligned. A linker can use this condition, when making the decision to generate an interworking veneer, to avoid the problem. There are two ways to do this:

1. If the linker can prove that the ARM target of a BLX instruction is not doubleword aligned, then it can use the BLX (immediate) instruction as normal.
2. The linker can generate veneers in such a way that no BLX (immediate) instruction branches to a doubleword aligned ARM target.

7.4.4.1 Prove that ARM target is not doubleword aligned

If the section that contains the ARM target is at least doubleword aligned and the ARM target is not doubleword aligned in that section then the ARM target is not doubleword aligned. The linker can use the BLX instruction as normal.

If all veneers that the linker can generate have a size that is a multiple of 8 bytes, and the section addresses are fixed after veneer generation, then an ARM target that is not doubleword aligned is not doubleword aligned after veneer generation. The linker can use the BLX (immediate) instruction as normal.

7.4.4.2 Generate veneers so that no BLX instruction branches to a doubleword aligned target

With only word alignment of sections, it is not sufficient to test the address of the ARM target to determine whether it is not doubleword aligned. Generated veneers must be added to the image, and this can change the base address of sections. This can affect the doubleword alignment of the ARM targets in the sections.

One way to avoid this problem is to iteratively convert BL instructions to BLX instructions in multiple passes. If the veneers added in pass N cause some ARM targets to be doubleword aligned, the linker must add new veneers on pass N + 1. The linker terminates the veneer generation process when there are no remaining doubleword aligned ARM targets.

7.4.4.3 Impact of exploiting increased section alignment

A section with doubleword alignment must be placed at the start of a doubleword aligned address. Given that sections contain word sized ARM instructions, on average 50% of these sections require an extra word of padding to make them start at a doubleword aligned address.

A section that contains ARM code might contain a number of ARM targets of BLX (immediate) instructions. On average, 50% of these ARM instructions are doubleword aligned. Therefore by making the section alignment doubleword aligned, 50% of ARM targets are not doubleword aligned.

In general, the linker can eliminate 50% of the veneers, for a cost of (0.5 * 4) bytes padding for each ARM section with a BLX target.

7.4.4.4 Special case of one function per section

Most toolchains have an option to compile each function in its own section. The majority of Thumb to ARM BLX instructions are function calls. Therefore, if each function is compiled in its own section:

- The vast majority of Thumb to ARM BLX targets are at offset 0 from the start of the section.
- There is only one ARM target in each section.

If the original section is only word aligned, the linker can add a word of padding to the start of the section. This causes the ARM target to start at a word offset from the start of the section.

Almost all veneers can be avoided, for a small cost in padding for each section containing an ARM target. This average cost is (0.5 * 4) bytes padding for the increased section alignment, and 4 bytes padding to prevent the ARM target from being doubleword aligned.

7.4.5 Additional optimizations

7.4.5.1 BLX to branch target

Condition 6 for the problem to occur, given in *Detecting the problem in a static toolchain*, is that the target of the BLX (immediate) instruction must not be an unconditional branch. If the ARM target instruction is a direct or indirect branch, then the toolchain can generate a BLX (immediate) instruction as normal. This applies to both workarounds.

7.4.5.2 Alignment of the Thumb BLX instruction

The problem is sensitive to the address of the BLX (immediate) instruction, providing that the BLX instruction is not itself a target of a branch. In theory, a linker can exploit this. However, doing so requires a more significant change than using the other workarounds. This subsection describes the required change.

The BLX must be congruent to 0 or 30 modulo 32 bytes. To exploit this information, the section alignment must be increased to 32. This requires, on average, the addition of 16 bytes of padding to each section, to enforce the alignment constraints.

It is cheap for a linker to check that a BLX is a target of a relocatable branch. However, unlike the BLX and BL instructions, the Thumb half-word unconditional and conditional branches do not usually have relocations. This is because the small immediate prevents any useful kind of redirection. A linker must disassemble the image to check for branch targets.

7.5 Procedure for generating state change veneers

This section contains a pseudocode description of the top level procedure for handling veneer generating relocations, as described in *ELF for the ARM Architecture*. It shows how a toolchain might use Veneer Generating Relocation Directives to identify and correct BLX instructions, at the same time as generating state change veneers.

```
For Each Veneer Generating Relocation Directive
    Find Caller State
    Find Callee State
    Find Range between Caller and Callee
    If (State Change Required)
        If Call Relocation, BLX Available, range < call range
            Write BLX at Relocation Place
        Else
            Generate a range extension state change veneer
    Else if (range < relocation range)
        Generate a range extension veneer
```