

Dynamic Linking with the ARM Compiler toolchain

Application Note 242



Dynamic Linking with the ARM Compiler toolchain

Application Note 242

Copyright © 2010 ARM. All rights reserved.

Release Information

The following changes have been made to this book.

Change History			
Date	Issue	Confidentiality	Change
December 2010	A	Non-Confidential	First release

Proprietary Notice

Words and logos marked with or are registered trademarks or trademarks of ARM in the EU and other countries, except as otherwise stated below in this proprietary notice. Other brands and names mentioned herein may be the trademarks of their respective owners.

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. ARM shall not be liable for any loss or damage arising from the use of any information in this document, or any error or omission in such information, or any incorrect use of the product.

Where the term ARM is used it means "ARM or any of its subsidiaries as appropriate".

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.

Web Address

<http://www.arm.com>

Contents

Dynamic Linking with the ARM Compiler toolchain

Application Note 242

Chapter 1	Conventions and feedback	
Chapter 2	Introduction	
	2.1 Purpose	2-2
	2.2 Additional material	2-3
Chapter 3	Linking overview and background	
	3.1 Linking background	3-2
	3.2 Symbol definitions and references	3-3
	3.3 Static linking and relocations	3-4
Chapter 4	How is dynamic linking different to static linking?	
	4.1 Dynamic linking concepts	4-3
	4.2 Dynamic linking benefits	4-4
	4.3 Disadvantages of dynamic linking	4-5
Chapter 5	Alternatives to dynamic linking	
	5.1 Symbol definition files	5-2
	5.2 Jump tables	5-3
	5.3 Overlays	5-4
Chapter 6	ELF overview and background	
	6.1 Introduction	6-2
	6.2 ELF file structure	6-3

Chapter 7	Technical overview of dynamic linking	
7.1	Dynamic linker	7-2
7.2	What does the dynamic linker need to do?	7-3
7.3	Locating dynamic content	7-5
7.4	Dynamic relocations	7-6
7.5	Handling function calls between modules	7-9
7.6	Handling data accesses between modules	7-13
Chapter 8	Controlling what symbols can be dynamically linked	
8.1	Symbol visibility	8-2
8.2	Populating the dynamic symbol table	8-4
8.3	Symbol preemption	8-5
8.4	Symbol versioning	8-8
Chapter 9	Dynamic linking models	
9.1	Bare-metal	9-2
9.2	DLL-like	9-3
9.3	SysV	9-5
9.4	Choosing an appropriate model	9-7
Chapter 10	Dynamic linking with the ARM Compiler toolchain	
10.1	Controlling symbol visibility in the ARM Compiler toolchain	10-2
10.2	Controlling dynamic section contents	10-7
	Glossary	

Chapter 1

Conventions and feedback

The following describes the typographical conventions and how to give feedback:

Typographical conventions

The following typographical conventions are used:

`monospace` Denotes text that can be entered at the keyboard, such as commands, file and program names, and source code.

`monospace` Denotes a permitted abbreviation for a command or option. The underlined text can be entered instead of the full command or option name.

monospace italic

Denotes arguments to commands and functions where the argument is to be replaced by a specific value.

`monospace bold`

Denotes language keywords when used outside example code.

italic Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.

bold Highlights interface elements, such as menu names. Also used for emphasis in descriptive lists, where appropriate, and for ARM® processor signal names.

Feedback on this product

If you have any comments and suggestions about this product, contact your supplier and give:

- your name and company

- the serial number of the product
- details of the release you are using
- details of the platform you are using, such as the hardware platform, operating system type and version
- a small standalone sample of code that reproduces the problem
- a clear explanation of what you expected to happen, and what actually happened
- the commands you used, including any command-line options
- sample output illustrating the problem
- the version string of the tools, including the version number and build numbers.

Feedback on content

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

- the title
- the number, ARM DAI 0242A
- if viewing online, the topic names to which your comments apply
- if viewing a PDF version of a document, the page numbers to which your comments apply
- a concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

ARM periodically provides updates and corrections to its documentation on the ARM Information Center, together with knowledge articles and *Frequently Asked Questions* (FAQs).

Other information

- *ARM Information Center*, <http://infocenter.arm.com/help/index.jsp>
- *ARM Technical Support Knowledge Articles*, <http://infocenter.arm.com/help/topic/com.arm.doc.faq>s
- *Support and Maintenance*, <http://www.arm.com/support/services/support-maintenance.php>
- *ELF for ARM Architecture*, http://infocenter.arm.com/help/topic/com.arm.doc.ihl0044d/IHL0044D_aa_elf.pdf.

Chapter 2

Introduction

Dynamic linking is a complex subject that is normally only partially understood. To understand dynamic linking in the ARM Compiler toolchain, you not only need a good dynamic linking background, but also some knowledge of the compiler-specific options relating to dynamic linking.

You often need to have a good understanding of the basics of dynamic linking to be able to understand the behavior of the tools. Providing this larger picture of dynamic linking is not practical on a case by case basis.

Information on dynamic linking is split into various places such as the ELF specification, the *Application Binary Interface* (ABI) for the ARM Architecture, Operating System platform specifications and the ARM Compiler documentation.

This application note provides an introduction to dynamic linking, describes the different linking models available and demonstrates how to build platform-specific images with the ARM Compiler. It is not intended to provide a deep, technical look at dynamic linking.

This section includes:

- [Purpose on page 2-2](#)
- [Additional material on page 2-3](#)

2.1 Purpose

The use of complex operating systems that use dynamic linking is increasing in systems designed for ARM processors. This means that there are an increasing number of engineers working on systems that use dynamic linking, for example, to generate dynamically loadable images.

This application note looks at the basic principles of how dynamic linking works and how the ARM Compiler toolchain can be used to generate various types of dynamic executables and shared libraries. The application note aims to help an applications developer to understand dynamic linking, the implications of dynamic linking, and the ARM Compiler toolchain dynamic linking related features.

The document also provides an overview of the different types of dynamically linked images that can be created. This might be useful as a starting point for initial investigations into implementing such a system, although this document does not provide a detailed look into specific implementation details.

Example code is provided in the *downloadable zip file*, [AN242_dynamic_linking_with_rvct.zip](#) to demonstrate how the ARM Compiler toolchain supports dynamic linking. ARM Compiler toolchain version 4.1, build 462 was used to generate the ELF files, which were processed to provide the disassembly listings found throughout the application note.

2.2 Additional material

This section lists publications by ARM and third parties.

See Infocenter, <http://infocenter.arm.com>, for access to ARM documentation.

ARM publications

The following documents contain information relevant to this document:

- *ARM Architecture Reference Manual (ARM ARM)*,
<http://infocenter.arm.com/help/topic/com.arm.doc.subset.architecture.reference/index.html#reference>
- *ARM Compiler toolchain documentation*,
<http://infocenter.arm.com/help/topic/com.arm.doc.subset.swdev.coretools/index.html>
 - Migration and Compatibility
 - Building Linux Applications with the ARM Compiler toolchain and GNU Libraries
 - Assembler Reference
 - Compiler Reference
 - Linker Reference
 - Using the fromelf Image Converter
 - Using the Compiler
 - Using the Assembler
 - Using the Linker.
- *Application Binary Interface (ABI) for the ARM Architecture*,
<http://infocenter.arm.com/help/topic/com.arm.doc.subset.swdev.abi>
 - ELF for the ARM Architecture
 - Base Platform ABI for the ARM Architecture.
- *What are Overlays and how are they used?*,
<http://infocenter.arm.com/help/topic/com.arm.doc.faqs/ka4234.html>

Other publications

The following documents list relevant documents published by third parties:

- *ELF Specification*, <http://www.sco.com/developers/gabi>

Examples Downloadable zip file, [AN242_dynamic_linking_with_rvct.zip](#).

Chapter 3

Linking overview and background

This section includes:

- *Linking background* on page 3-2
- *Symbol definitions and references* on page 3-3
- *Static linking and relocations* on page 3-4.

3.1 Linking background

Before looking at dynamic linking it is important that basic linker terminology and behavior is understood. A dynamic linker performs a number of similar operations to a static linker. Therefore, the first thing covered in this application note is some basic information on what a linker does and how it does it.

At a high level a linker takes a number of objects files, normally produced by a compiler or assembler, and combines them into an executable file. Some of the object files included in the image might be stored in libraries.

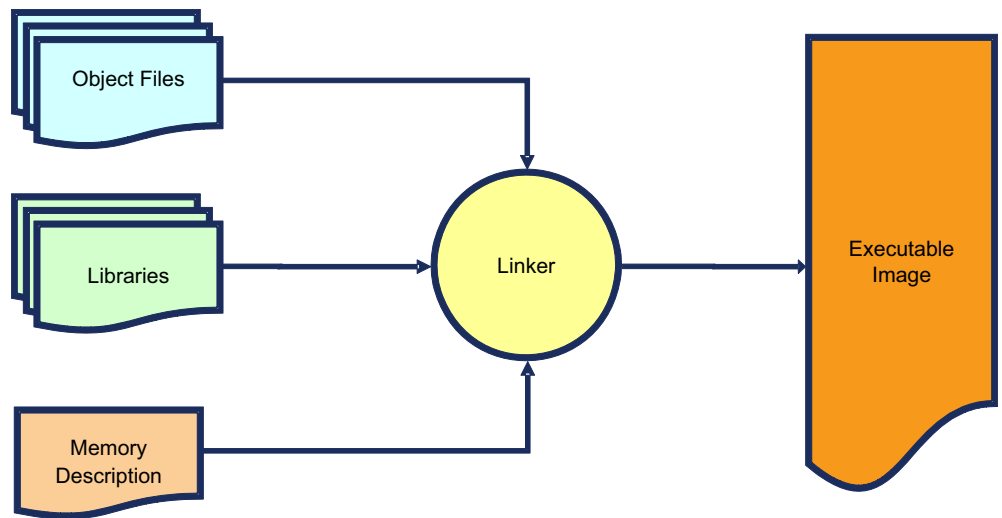


Figure 3-1 Linking process

The linker also needs information on the memory layout for the executable image. This can be provided in a number of ways, including linker command-line options, memory (scatter-loading) description files or default values within the linker.

The linker is responsible for locating individual parts of the object files in the executable image, ensuring that all the required code and data is available to the image and any addresses required are handled correctly.

3.2 Symbol definitions and references

To ensure that an image includes all the required code and data, the linker must bind the unresolved symbol references in the input object files to definitions. When linking, it is normal to refer to a symbol rather than specifically to whether something is a function or variable.

A *definition* is where the linker loads the code for a function or variable. A *reference* is where a symbol is used, for example a function call or using a global variable. Take the following C code examples:

Example 3-1

```
extern void bar(void);

void foo(void)
{
    bar();
}
```

In this example the function foo is a definition and bar is a reference.

Example 3-2

```
int var;

int value(void)
{
    return var;
}
```

Here var and value are definitions.

During the linking process a linker must match a single definition to every reference. In [Example 3-2](#) only a definition of var is required in the object file because var is defined in the same source file where it is used. In [Example 3-1](#) the linker requires a definition of bar from another object file to resolve the reference to bar.

3.3 Static linking and relocations

An object producer, normally a compiler or assembler, only has visibility of the source code it is processing and has very limited information about the image layout. In [Example 3-1 on page 3-3](#) of section: [Symbol definitions and references on page 3-3](#), the object producer (C/C++ compiler) knows that it needs to call the bar function using some form of branch instruction, but it has no knowledge about the address of bar used in the final executable image. Therefore, the compiler cannot generate the offset or address for the branch instruction.

To solve this problem the object producer generates a *relocation* to instruct the linker to fill in the required offset or address.

If you are already familiar with relocations and how they work, you can skip the following section, but if this is not the case, or you want to refresh your existing knowledge, go to the `examples/example_3-1` directory to see how this works with the ARM Compiler toolchain. This directory contains the C source code for [Example 3-3](#) and a build script, which when run, displays the code and relocation information for the ELF object.

Section 4.6.1.2 - "Relocation types" from the *ELF for the ARM Architecture ABI* document, can be used to check the type of relocation generated for each example. The following output is generated by the `fromelf` utility of the ARM Compiler toolchain. The `fromelf` utility is an ELF reader and image converter.

Example 3-3 Sample output from `fromelf -c -r foo.o`

```

** Section #1 '.text' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
  Size :    4 bytes (alignment 4)
  Address: 0x00000000

  $a
  .text
  foo
    0x00000000:  EFFFFFFE    ....    B            bar

** Section #6 '.rel.text' (SHT_REL)
  Size : 8 bytes (alignment 4)
  Symbol table #5 '.symtab'
  1 relocations applied to section #1 '.text'
  #   Offset      Relocation Type   Wrt   Symbol   Defined in
  0   0x00000000    29 R_ARM_JUMP24    7     bar     Ref

```

This output shows a single jump relocation with a relocation type 29, `R_ARM_JUMP24`. The offset of the relocation is `0x0` bytes into the file, and the relocation is for the reference to the symbol `bar`, which is the seventh symbol listed in the symbol table of the object.

The operation for the `R_ARM_JUMP24` relocation is: $((S + A) \mid T) - P$.

S is the address of the symbol, which equals `0x8000` if you check the address information of the final image.

A is the addend for the relocation. The addend for this "REL type relocation" is:

```

sign_extend (insn[23:0] << 2) = sign_extend (0xFFFFFE << 2)
                                = sign_extend 0xFFFFF8
                                = 0xFFFFFFF8
                                = -0x81

```

The ELF for ARM Architecture document states that T is 1 if the target symbol has global binding and addresses a Thumb instruction. In the `example_3-1` directory a definition for the symbol `bar` exists in a separate unit, and the build script targets that unit for ARM code. Therefore T is 0.

P is the address of the place being relocated, which equals 0x8018.

$$\begin{aligned} ((S + A) \mid T) - P &= (0x8000 + -0x8) \mid 0 - 0x8018 \\ &= 0xFFFFFE0 \\ &= -0x20 \end{aligned}$$

Therefore, an offset of -0x20 is required for the branch instruction in [Example 3-3 on page 3-4](#).

The `fromElf` disassembly listing for the final executable image shows that the opcode used for the branch instruction is 0xEAFFFFF8. This opcode has been inserted by the static linker to ensure that the instruction causes the processor program counter to branch to `bar`. But why has the opcode changed from 0xEAFFFFFE (as seen in the object file) to 0xEAFFFFF8, as shown in the following example?

```
** Section #1 'ER_R0' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
   Size : 28 bytes (alignment 4)
   Address: 0x00008000
   $a
   .text
   bar
       0x00008000: E59f000C .... LDR r0,[pc,#12] ; [0x8014] = 0x801C
       0x00008004: E5901000 .... LDR r1,[r0,#0]
       0x00008008: E2411001 ..A. SUB r1,r1,#1
       0x0000800C: E5801000 .... STR r1,[r0,#0]
       0x00008010: E12FFF1E ../. BX lr
   $d
       0x00008014: 0000801C .... DCD 32796
   $a
   .text
   foo
       0x00008018: EAFFFFF8 .... B bar ; 0x8000
```

Under normal circumstances it is not necessary to understand the exact syntax of a specific instruction, but sometimes this information is of interest, for example, when debugging or trying to understand how instructions and relocations work. Also, a linker is responsible for modifying or inserting instructions into an executable image to ensure that it runs correctly, so it is still important to have a good understanding of how an instruction is formed. To explain what opcode 0xEAFFFFF8 means, the full syntax of a branch instruction is shown in [Figure 3-2 on page 3-7](#). This is taken directly from the *ARM Architecture Reference Manual*.

Bits 31-28 of the opcode hold the condition under which the instruction is executed, denoted by 0xE, which is 1110 in binary. This is the AL condition that means always execute or execute unconditionally.¹

1. The value 0xFFFFFFF8 is negative, but it is difficult for the human eye to read. One of the best ways to find out what negative number this value represents is to invert all of the bits and add 1 to the result (twos complement) giving: 0x00000007 + 1, which equals 0x8. Therefore the negative number 0xFFFFFFF8 represents is: -0x8. For more information, see [Figure 3-2 on page 3-7](#) for a description of the branch and branch with link instruction, and also [Table 4-11](#), ARM relocation actions by instruction type, from the *ELF for ARM Architecture*, http://infocenter.arm.com/help/topic/com.arm.doc.ih0044d/IHI0044D_aaelf.pdf.
1. It is not necessary to specify the AL condition when writing ARM assembler language which is why it is not displayed in the `fromElf` disassembly listing.

Bits 27-25 are always set to 101 for a branch (B) or branch with link (BL) instruction. Bit 24 is not set, so this means that the instruction is a standard branch (B), which also means that the processor does not store a return address in the link register, R14, because `bar` does not return. Therefore, the hexadecimal number denoted by the `0xA` is stored into bits 27-24 of the branch instruction.

The other bits, bits 23-0, are used to specify the target address of the ARM branch instruction. The following steps are required to calculate the target address:

1. Sign extend the 24-bit signed immediate to 30 bits, so the value `0xFFFF8` becomes `0x3FFFFFF8`.
2. Shift the resulting value left by two bits to form a 32-bit value of `0xFFFFFE0` (offset `-0x20` calculated above, using twos complement).
3. Add the value to the contents of the PC, which contains the address of the instruction plus 8 bytes, because of the processor instruction pipeline. Adding `0x8` (8 bytes) to `0x00008018` (the address of the branch instruction) gives: `0x00008020`.

$$0x00008020 + -0x00000020 = 0x00008000.$$

Example 3-4 Sample output from `fromelf -r bar.o`

```

** Section #7 '.rel.text' (SHT_REL)
   Size : 8 bytes (alignment 4)
   Symbol table #6 '.symtab'
   1 relocations applied to section #1 '.text'

# Offset      Relocation Type  Wrt  Symbol  Defined in
0 0x0000000C   2 R_ARM_ABS32      4    .data   #4 '.data'
```

This output shows a single 32-bit absolute relocation in the above output with a relocation type 2, `R_ARM_ABS32`. The offset of the relocation is `0xC` bytes into the file and the relocation is for the reference to the symbol `var`. The symbol `var` is defined in a read/write data section (#4) called `.data`, which is the fourth symbol listed in the symbol table of the object.

The operation for the `R_ARM_ABS32` relocation is: $(S + A) \mid T$.

S is `0x8010`, if you check the address information of the symbol `var` in the final image.

A (the addend) is 0, because the value at address `0x800C` is `0x0` in the ELF object.

T is also 0 because the target symbol addresses an ARM instruction.

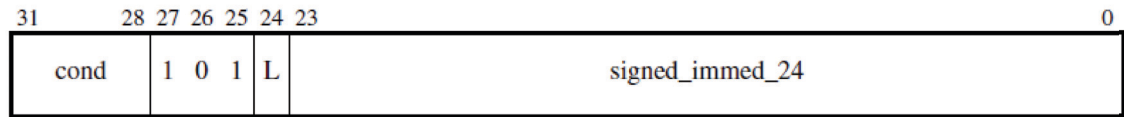
The result is:

$$(S + A) \mid T = (0x8010 + 0x0) \mid 0) \\ = 0x8010$$

Therefore, the value `0x8010` (the address of `var`) is filled into the literal pool.

ARM Instructions

A4.1.5 B, BL



B (Branch) and BL (Branch and Link) cause a branch to a target address, and provide both conditional and unconditional changes to program flow.

BL also stores a return address in the link register, R14 (also known as LR).

Syntax

B{L}{<cond>} <target_address>

where:

- L Causes the L bit (bit 24) in the instruction to be set to 1. The resulting instruction stores a return address in the link register (R14). If L is omitted, the L bit is 0 and the instruction simply branches without storing a return address.
- <cond> Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If <cond> is omitted, the AL (always) condition is used.

<target_address>

Specifies the address to branch to. The branch target address is calculated by:

1. Sign-extending the 24-bit signed (two's complement) immediate to 30 bits.
2. Shifting the result left two bits to form a 32-bit value.
3. Adding this to the contents of the PC, which contains the address of the branch instruction plus 8 bytes.

The instruction can therefore specify a branch of approximately $\pm 32\text{MB}$ (see *Usage* on page A4-11 for precise range).

Architecture version

All.

Figure 3-2 Extract from the ARM Architecture Reference Manual (ARM ARM)

Chapter 4

How is dynamic linking different to static linking?

To successfully create a statically linked image, the linker must ensure that the image produced contains all the functions and variables required for that image by resolving every reference to a definition. The linker has information on the memory layout. Therefore it can locate all the code and data as required. This not only allows the linker to ensure that all the required code and data is linked into the image, but also allows it to resolve all the relocations in the image. The final image that is produced can then be loaded directly into memory and executed without any need for further processing.

However, in a dynamically linked environment, the code and data spreads over a number of different modules. For a particular application to execute, it might have to use the code and data from one or more of these modules.

The execution addresses of the various modules are not known until they are loaded onto the platform. This means the static linker cannot resolve the references and relocations between different modules. Therefore, before the image is executed on the platform, a further link step is required to handle the references and relocations between the different modules/images. This link step is known as dynamic linking.

The job of both the static and dynamic linker is to bind the symbolic references present in code and data to real addresses that can be executed at run-time. Binding symbolic references at static link time is typically efficient at run-time, but inflexible. Binding symbolic references at dynamic linking is more flexible but less efficient.

There are a number of places that the dividing line between the programs can be drawn. At one extreme we have all static linker and no dynamic linker when there is no run-time loading of code. At the other extreme there is no static linker and every symbolic reference is bound by the dynamic linker, either at load-time or even on demand. However, the majority of platforms have both a static and dynamic linker to allow a blend of flexibility and efficiency.

Another way of thinking about it is that dynamic linking is the completion of decisions deferred from static link time. The static linker has to encode enough information in the dynamic segment for the dynamic linker to resolve these decisions.

This section includes:

- [*Dynamic linking concepts*](#) on page 4-3
- [*Dynamic linking benefits*](#) on page 4-4
- [*Disadvantages of dynamic linking*](#) on page 4-5.

4.1 Dynamic linking concepts

A dynamic application normally consists of an executable and a number of modules. The modules are normally referred to as either a shared library, *dynamic link library* (DLL) or *dynamic shared object* (DSO), depending on the operating system. A module might be required by the application or by one of the modules used by the application. [Figure 4-1](#) shows a simple example of this.

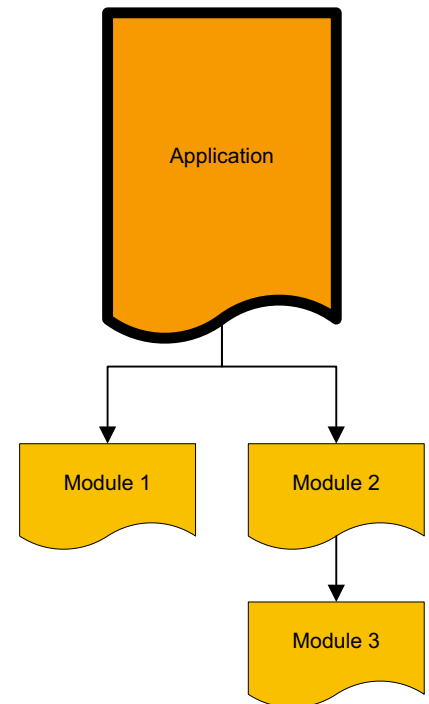


Figure 4-1 Simple dynamic application

When the application is executed a dynamic linker/loader, referred to as the dynamic linker throughout this document, loads the application into memory. The dynamic linker must also load any modules required by the application and any other required modules.

Any module required by an application or a different module is called a dependency. The dynamic linker typically needs to know the dependencies for the application and every module so that it can load all the required modules¹. The dynamic linker also needs to ensure that any function calls or data accesses between the application and modules functions correctly.

To do this the dynamic linker must know what functions and data are available in a particular module. The dynamic linker also needs to know where to handle calls and data accesses between the different modules.

This means that the applications and modules need extra information encoded into the files to allow the dynamic linker to function correctly. This is described in [Chapter 7 Technical overview of dynamic linking](#).

1. Programs can use system functions such as the UNIX commands `dlopen` and `dlsym` to load an arbitrary shared library at run-time. In this case `dlopen` is explicitly naming the dependency. The `dlsym` function names the function definition in the library.

4.2 Dynamic linking benefits

There are two main advantages to dynamic linking over static linking:

- Shared library updates
- Potential memory usage savings.

Shared libraries can be updated. For example, a media player application might originally be shipped with a codec that supports the mp3 file format. If the media player were statically linked it would not be possible to dynamically update it to support a different file format, without replacing the entire application. Dynamic linking means that a new version of the shared library containing a more up-to-date codec, which includes some enhancements and bug fixes, could be dynamically loaded by a dynamic linker into memory at run-time to replace the original shared library.

A shared library can also be shared by more than one application. For example, two different media players could both use the same shared library containing the same codec. This potentially means that the device running the application requires less physical memory, depending on the size of the dynamic linker.

4.3 Disadvantages of dynamic linking

A dynamically linked system has some run-time overhead. For example, calling functions through a *Procedure Linkage Table* (described in [Handling function calls between modules on page 7-9](#)). There is also the complex task of creating a dynamic linker. This can be very time consuming, and a dynamic linker can typically require maintenance. Therefore, give serious consideration to whether implementing a full dynamic linking system is the appropriate solution.

In a lot of cases static linking is not possible, for example, because of limited memory, and a full dynamic linking system is also undesirable due to the complications mentioned previously. However, the requirements for a particular application can still be met. Such applications can use alternative features of the ARM Compiler toolchain, which lie somewhere between static linking and a full dynamically linked system.

There are a number of common approaches. These include using *symbol definition files*, *jump tables* and *memory overlays*.

Chapter 5

Alternatives to dynamic linking

There are alternatives to complying with one of the dynamic linking models described in [Chapter 9 *Dynamic linking models*](#). Some of these are listed in the following sections. These options permit more bespoke dynamic linking models, which are typically simpler and require little or no knowledge of dynamic linking information within ELF files.

This section includes:

- [Symbol definition files on page 5-2](#)
- [Jump tables on page 5-3](#)
- [Overlays on page 5-4.](#)

5.1 Symbol definition files

A *symbol definition file* (or *symdefs* file) is an optional output file from a static linker that lists the memory addresses of symbols for the generated image. This can then be used as an input to a later link step. The static linker can resolve references to symbols contained in the symbol definition file using the address provided in the symbol definition file.

A symbol definition file allows an application to be split into a number of separate pieces but still permits function calls and data accesses between them. For example, an application could be broken into a core application and a number of modules. Each module could be created separately and then the symbol definition file can be used when linking the core application. Alternatively, the symdefs file of the core application can be used when developing the modules. In effect the symdefs file contains the addresses of entry points of the core application.

Because symdefs files contain the addresses of symbols, they must be used with caution. Users must ensure that any symdefs file used is correct and corresponds exactly with the binary files that are to be used on the target hardware. If there are any mismatches between the symdefs file and binaries on the target hardware, it is likely to result in a link-time error or run-time failure.

5.2 Jump tables

A jump table (or branch table) is a method used to remove dependencies between different versions of binary files. The simplest form of a jump table is a table of branch instructions. When calling a function in a separate binary file, the code branches into the jump table, and the jump table then branches to the required function.

Each function in a binary file that can be called from another binary has an entry in the jump table. The jump table is at a fixed location that is not changed between different versions of the binary file. Therefore any calls to a particular jump table entry results in the correct function being executed. The exact location of the function does not matter to the calling application and can vary between versions of the binary file.

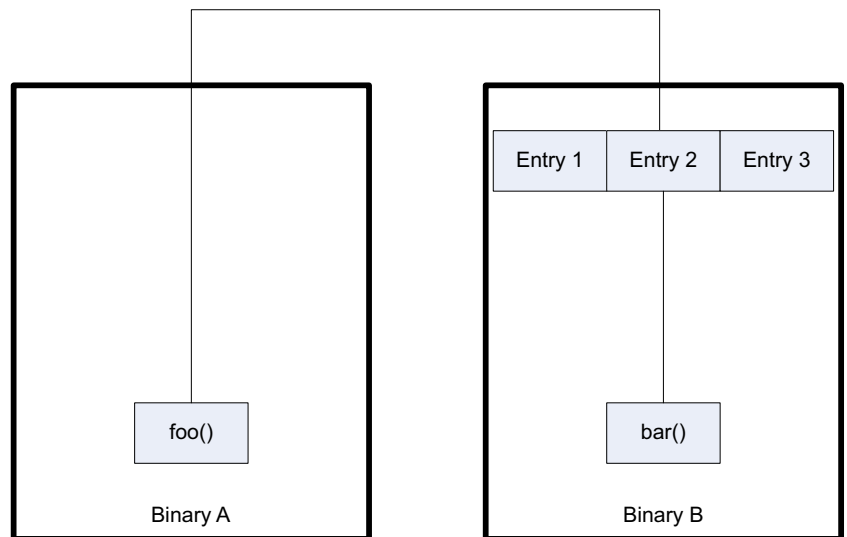


Figure 5-1 Jump table

The example in [Figure 5-1](#) shows two binaries. Binary A contains a function called `foo` that calls a function called `bar` in Binary B, using the second entry in a three-entry jump table. Each time a new version of the Binary B is created, provided that the address of the jump table is statically linked to the same location, the static linker does not need to worry about the address it gives to function `bar`. This is because any calls to `bar` go through the jump table.

In the example Binary A has no jump table, but if `bar` called `foo`, and Binary A is likely to be replaced in the future with a newer version, then Binary A also requires a jump table.

A situation where jump tables might be useful is when a system includes an RTOS, which is delivered with lower level functions for handling input and output. The addresses of each function are stored in a given entry of the jump table (table of function pointers). If the RTOS is updated, some of the lower level functions might move to new locations. Any threads from a different binary which rely on the functions can still branch to them without the threads needing to be rebuilt and reloaded.

Jump tables are normally written in assembler and then placed at a fixed location in memory. It is critical that this location is not modified and that the location of individual jump table entries does not change between builds.

A `symdefs` file that only includes entries for the jump table can be used when generating other applications or binaries that use the functions.

5.3 Overlays

Overlays permit multiple sections of code and data to be linked to execute from the same address within the same image. Normally in a statically linked image code and data is linked at a unique location and the static linker does not allow multiple items to be placed at the same location.

An executing image that uses overlays has to manage which code and data, normally referred to as an overlay or overlay region, is currently in the shared memory location. The image has to ensure that the correct overlay is active (in memory) when trying to execute code or access data from within it.

An image normally has an overlay manager that handles which overlay is currently active and copies the code and data to the overlay memory region.

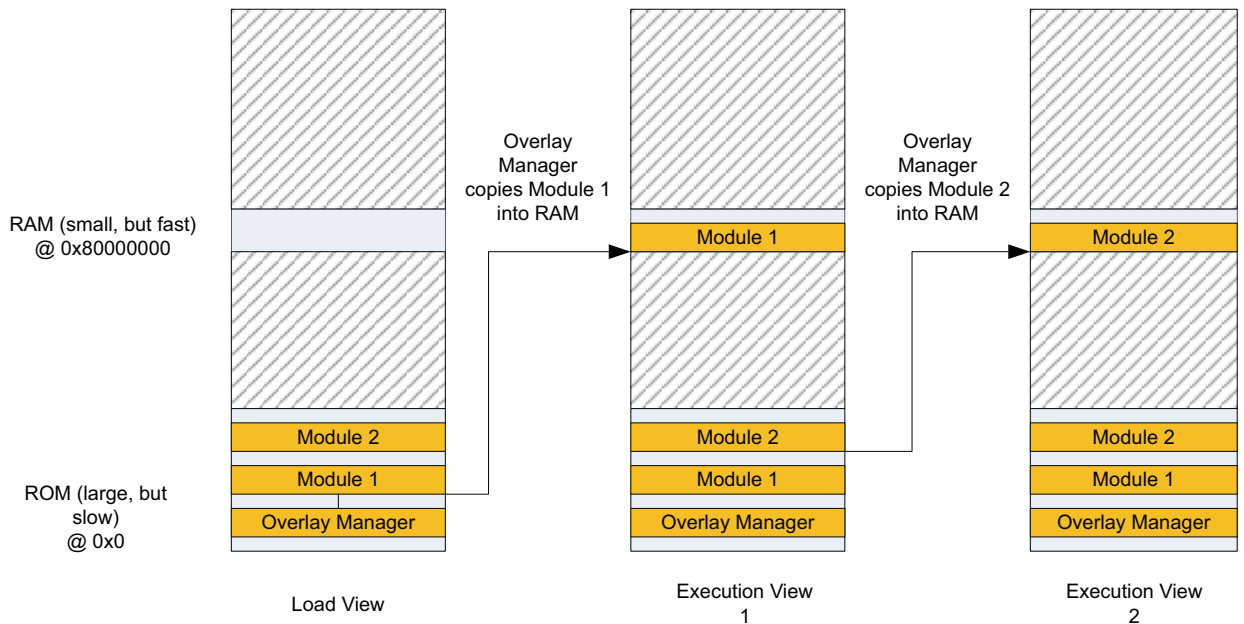


Figure 5-2 Overlays

Chapter 6

ELF overview and background

This section includes:

- *Introduction on page 6-2*
- *ELF file structure on page 6-3.*

6.1 Introduction

The ARM Compiler toolchain use the ELF file format for object files and executables. This is a standard file format used by a large number of tools. A number of operating systems execute ELF files directly, but for some operating systems, the ELF file may need to be converted into another specific file format.

Note

The ELF file format is described in the ELF specification. However, some ELF file features are ARM specific and this additional information can be found in the ARM ELF ABI documentation.

An ELF file contains a number of different sections. Each section holds a block of data of a specific type. The specific type could be program code or data, but it might also be non-program information, such as debug data.

The ELF file also includes data structures that describe the contents of the ELF file. These include an ELF header, a section header table and a program header table. [Figure 6-1](#) shows a top-level view of an ELF file.

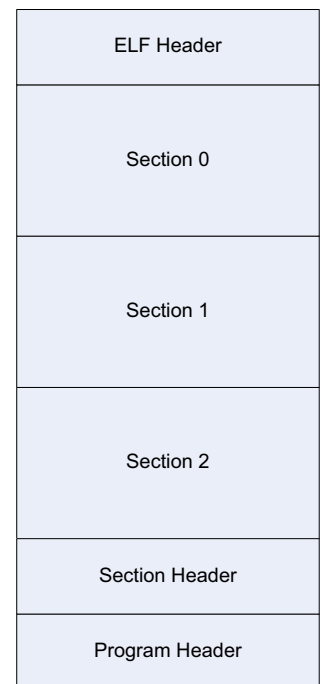


Figure 6-1 Basic view of an ELF file

Note

The section header table and program header table are normally placed at the end of the ELF file but this is not guaranteed by the ELF specification.

6.2 ELF file structure

This section includes:

- [ELF headers](#)
- [ELF sections](#).

6.2.1 ELF headers

ELF header

The ELF Header contains information about the ELF file. For example, it identifies the file as an ELF file, states the hardware architecture the ELF file is built for (for example, EM_ARM, which stands for "ELF Machine: ARM"), and gives the locations of the section and program header tables.

Program header

The program header table contains entries for one or more program headers. Program headers describe a continuous range of bytes in the ELF file. The program header includes information on the segment type, properties, and address information.

A program header does not just describe executable code or data. That is true only for a program header of type PT_LOAD. Also, not every section in the image has an associated program header. An image loader or operating system normally reads the program header to locate the required information to prepare an ELF file for execution.

An important type of program header for dynamic linking has a type PT_DYNAMIC, that describes the location of the dynamic section.

Section header

The section header table is a table that describes each of the sections in the ELF file. Each section in the ELF file has an entry in the section header table. A section header table entry contains the section name, section type, flags (for example, read/write), execution address, file offset, and other information.

For ELF images, the section header table is optional. A dynamic loader that works directly on ELF files should not rely on information derived from section headers. As an example the dynamic linker can locate the dynamic section by using the PT_DYNAMIC program header, or by looking for the dynamic section in the section header table. The former works if the section header table is removed, the latter does not.

6.2.2 ELF sections

An ELF file normally has a number of standard sections. A section is block of similar data. An ELF file normally comprises sections containing program code, program data, dynamic information, relocations, a symbol table, and a string table. Each section in the file has a section header table entry describing the section and its properties.

The following sub-sections provide an overview of the most commonly found ELF sections.

Program sections

An ELF file normally includes multiple program sections. There is normally a program section for read only code, *read/write* (RW) data and *zero-initialized* (ZI) data. It is also common to have a section for read only data.

These sections normally have standard names:

- `.text` for executable code
- `.data` for RW data
- `.bss` for ZI data.

The string table

The string table is a section containing a list of null terminated strings. These are referenced from elsewhere in the ELF file, for example, the symbol table, using an offset into the string table.

For dynamically linked images there are normally two string tables:

- `.dynstrtab` for strings relating to dynamically linked symbols
- `.strtab` for statically linked symbols.

The symbol table

The symbol table describes the symbol definitions and references in the ELF file. It holds the address, binding, visibility, section, symbol type (code/data), size and name (via an offset into the string table) of each symbol.

In a dynamic image there might be two symbol tables:

- `.symtab` contains the static symbol table
- `.dynsym` contains the dynamic symbol table.

Note

It is common to remove the static symbol table from images to save space and hide IP.

The dynamic section

The dynamic section, normally named `.dynamic`, is used to provide specific information to the dynamic linker. This is achieved through entries called tags. There are a number of different types of tags. Each tag is used to provide a specific piece of information to the dynamic linker. Each tag has a value or associated piece of data.

For example `DT_NEEDED` tags are used to inform the dynamic linker which modules (dependencies) the current image requires. There are also tags to inform the dynamic linker of the location of certain items in the ELF file.

The tags in the dynamic section tell the dynamic linker where to find all of the other data, such as the dynamic symbol table, needed by the dynamic linker.

It is important for a dynamic linker to use the tags in the dynamic section to find the dynamic content, rather than use the section header table.

Relocation sections

Relocation sections are used to store the relocations in the ELF file.

For object files these sections are normally named `.rel<name>` or `.rela<name>`, where `<name>` is the section that the relocations apply to.

- `.relname` is used for `rel` types relocations
- `.relaname` is used for `rela` type relocations.

For a dynamically linked ELF file, the section is normally named `.dyn`.

Chapter 7

Technical overview of dynamic linking

This section includes:

- *Dynamic linker on page 7-2*
- *What does the dynamic linker need to do? on page 7-3*
- *Locating dynamic content on page 7-5*
- *Dynamic relocations on page 7-6*
- *Handling function calls between modules on page 7-9*
- *Handling data accesses between modules on page 7-13.*

7.1 Dynamic linker

A dynamic linker is platform-specific and is often part of the operating system. As previously mentioned the dynamic linker requires various pieces of information to allow it to prepare an image to be executed on a particular platform. This information is included in the executable file along with the code and data.

The actual format of the image loaded onto the platform is specified by the particular operating system. This section looks at how dynamic information is stored in ELF files. ELF is the file format used by a number of common operating systems and operating systems with custom file formats often convert an ELF file to an operating system specific format.

In an ELF file the information required by the dynamic linker is split up into the following locations:

- The dynamic section
- The dynamic relocation section
- The dynamic symbol table
- The dynamic string table.

[Chapter 6 *ELF overview and background*](#) describes these sections in more detail.

7.2 What does the dynamic linker need to do?

This section summarizes the actions that the dynamic linker or operating system has to do. It includes a rough walk through of the following steps carried out by the dynamic linker or operating system.

1. Invoke the dynamic linker.
2. Read any dependencies.
3. Load all files into memory.
4. Resolve (dynamic) relocations.

A dynamic linker is normally closely integrated into an operating system and the exact tasks it has to perform depends on the operating system. However the basic operations and the order these operations are performed in are normally very similar.

Note

The dynamic linker might optimize a number of these steps to improve startup times for applications and avoid unnecessary work.

When an application is executed that has one or more dependencies, the dynamic linker is called. In many cases there is only a single dynamic linker available in an operating system and this is called automatically. However, on some operating systems it is possible to have more than one dynamic linker and the operating system selects the appropriate dynamic linker based on information in the executable image. It is also possible on some systems to debug and trace the execution of an application and its dependencies. For example, under Linux it is possible to set the LD_DEBUG environment variable to output information from the dynamic linker.

When the dynamic linker is invoked, one of the first actions it performs is to process the dependency list for the executable image. The dependencies are the other executable modules that the executable image needs to execute. The dynamic linker must load these modules. It then processes the dependencies of the loaded modules. This process continues until all the dependencies of each module have been processed by the dynamic linker.

When all the dependencies are resolved, the dynamic linker can load the executable and all the required modules into memory. Again the load address for each executable and module is operating system specific.

When all the executable and modules are loaded into memory the dynamic linker can process the dynamic relocations in the application and modules. This includes the relocations against the PLT and the GOT.

When the relocations have been processed the executable image is ready to be executed. Therefore control passes to the application by branching to the entry point of the application.

7.2.1 BPABI and post-linking

It is important to mention that the ELF files produced by the BPABI are not designed to be loaded directly by a dynamic linker. However, it may be possible to do so in some cases, for example, by using `--pltgot=direct` or `--pltgot=none`.

The BPABI model is designed to produce ELF files compatible with the *Base Platform ABI for the ARM architecture* (BPABI). One of the assumptions underpinning the BPABI is that many platforms consider ELF too large and bulky a format to have on an embedded device. The ELF file produced by the BPABI is therefore designed to be post-processed by a post-linker to produce the file format for the platform OS.

An example of this is the Symbian elf2e32 converter. This takes the maximally symbolic ELF file, strips out the symbol information and replaces it with address tables indexed by ordinals and finally outputs a custom, compressed, signed e32 file.

This is the reason why `--pltgot=indirect` does not create the GOT. The intention is that a post-linker creates the GOT in a non-ELF format.

7.3 Locating dynamic content

One of the first tasks a dynamic linker has to perform when loading an ELF file is to find the dynamic content.

In an ELF file produced by the ARM compiler the location of the dynamic content is given by the corresponding section header table entry, and by the tags in the `.dynamic` section.

If the dynamic linker is loading ELF files directly then it should be using tags in the `.dynamic` section to find the dynamic content. This is because the presence of the section header table in an ELF executable is optional.

To find the dynamic content using the `.dynamic` section table the following process is used:

1. Load the program header table and find the table entry with type `PT_DYNAMIC`. This gives the location and size of the dynamic section.
2. Load the contents of the dynamic section using the information given by the `PT_DYNAMIC` program header table.
3. The contents of the dynamic section is a table of tag and value pairs. The location of the dynamic content is provided by several table entries. For example the dynamic string table is described by:

Table 7-1 Dynamic section content

Tag	Value
DT_STRTAB	Base of dynamic string table
DT_STRABSZ	Size in bytes of the dynamic string table

Note

When using the SysV dynamic linking model, which assumes that the dynamic linker loads ELF files directly, all values that are base addresses are virtual addresses.

When using the BPABI and base platform dynamic linking models, which assume that the ELF file is post processed by a platform specific post-linker, all values that are base addresses are file offsets.

7.4 Dynamic relocations

There are many different types of relocations. The majority of relocations are resolved at static link time by a static linker such as `arm1ink`. When an application uses shared libraries, it cannot know at static link time where the shared libraries are going to exist in memory. Therefore, dynamic relocations are added to such an application so that the dynamic linker can resolve the address information for each dynamically referenced symbol at load or run-time.

Each type of dynamic relocation informs the dynamic linker what type of address needs to be filled in, such as an absolute address or an offset, and the calculation to be used to generate the value. A relocation entry includes the target symbol, the type of relocation and an addendum.

The addendum is a value passed from the object producer to the dynamic linker to be used in the calculation. An addendum is normally used to provide an offset into a structure or to correct for PC bias. There are two methods used to store the addend:

- the more commonly used `rel` (`SHT_REL`) type relocations where the addendum is encoded into the address space which the dynamic linker must overwrite
- the less common `rela` (`SHT_RELA`) relocation type, where the addendum is encoded into the relocation.

7.4.1 BPABI relocation example

Example 7-1 C source file, `foo.c` with a reference to `bar` and `x`

```
__declspec(dllimport) void bar(void);
__declspec(dllimport) extern int x;

int foo(void)
{
    bar();
    return x;
}
```

Example 7-2 Shared library, `shared.c` defining `bar` and `x`

```
__declspec(dllexport) int x = 1;

__declspec(dllexport) void bar(void)
{
    x++;
}
```

In [Example 7-1](#) there is a reference to a function called `bar` and a variable called `x`. Both symbols are marked as being defined in another module with the `__declspec(dllimport)` attribute. When the ELF object, generated from this example code, is statically linked with the shared library ([Example 7-2](#)) that defines `bar` and `x`, a dynamic relocation section, `.dyn`, is generated in the executable image, for example:

```

** Section #5 '.dyn' (SHT_REL)
   Size   : 16 bytes (alignment 4)
   Symbol table #3 '.dynsym'
   2 relocations applied to section #0 '[Anonymous Section]'

```

#	Offset	Relocation Type	Wrt	Symbol	Defined in
0	0x0000800C	2 R_ARM_ABS32	2	bar	Ref
1	0x00008024	2 R_ARM_ABS32	3	x	Ref

The relocation type added to the image is R_ARM_ABS32 (relocation number 2 from section 4.6.18 of the ABI ELF for the ARM Architecture document). This is an ARM absolute 32-bit relocation, which is commonly found in static and dynamic relocation sections. To get a better feel for relocations, it is a good idea to have read the ELF for the ARM Architecture document.

At load or run-time the dynamic linker might load the shared library into memory, if it is required by the application or another shared library (module). When the dynamic linker has loaded the shared library, it has to know the addresses given to symbols x and bar. Using this information, the dynamic linker then needs to resolve all dynamic relocations to these symbols from the executable image, or any other application or module that requires this shared library.

"Table 4-8, Relocation codes" from the ELF for the ARM Architecture document shows the operation required to calculate the address used for the R_ARM_ABS32 type relocation:

$(S + A) \mid T$

S = the address of the S

A = the addend for the relocation

T = 1 if the target symbol S has type STT_FUNC and the symbol addresses is a Thumb instruction; it is 0 otherwise.

In [Example 7-1 on page 7-6](#), the symbol x does not have type STT_FUNC, and neither is it the symbol address of a Thumb function. Therefore, T = 0.

The relocations in the image are of type SHT_REL, so the addend for the relocation is encoded into the address which the dynamic linker needs to fill in to resolve the relocation at run-time. This is address 0x00008024, which has a value of 0x00000000 in the executable image:

```

** Section #1 'ER_RO' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
   Size   : 40 bytes (alignment 4)
   Address: 0x00008000

:
:
$a
.text
foo
    0x00008010: E92D4010 .@-. PUSH {r4,lr}
    0x00008014: EBFFFFFF .... BL 0x8004 ; 0x8004
    0x00008018: E59F0004 .... LDR r0,[pc,#4] ; [0x8024] = 0
    0x0000801C: E5900000 .... LDR r0,[r0,#0]
    0x00008020: E8BD8010 .... POP {r4,pc}
$d
    0x00008024: 00000000 .... DCD 0

```

Therefore, there is no addend; A is 0x0.

The following disassembly listing shows the code and data sections of the shared library:

```

** Section #1 'ER_R0' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
   Size   : 24 bytes (alignment 4)
   Address: 0x00008000

   $a
   .text
   bar
      0x00008000: E59F000C .... LDR    r0,[pc,#12] ; [0x8014] = 0
      0x00008004: E5901000 .... LDR    r1,[r0,#0]
      0x00008008: E2811001 .... ADD    r1,r1,#1
      0x0000800C: E5801000 .... STR    r1,[r0,#0]
      0x00008010: E12FFF1E ../.  BX     lr
   $d
      0x00008014: 00000000 .... DCD    0

** Section #2 'ER_RW' (SHT_PROGBITS) [SHF_ALLOC + SHF_WRITE]
   Size   : 4 bytes (alignment 4)
   Address: 0x00000000

      0x00000000: 01 00 00 00 .....

```

———— Note ————

Although the `fromElf` output shows `ER_R0` starting at address `0x8000`, it can be relocated or loaded to a different address such as `0x9000` by the dynamic linker.

Suppose the dynamic linker loads the shared object (module) that contains the symbol `x` to address `0x9000`. This means that the first instruction in the code section (#1) starts at address `0x9000` and the last instruction is at address `0x9014`.

Assuming the dynamic linker loads the data section (#2) of the shared library directly after the code section, the address of symbol `x` is `0x9018`, and the following calculation can be applied:

$$\begin{aligned}
 & (S + A) \mid T \\
 &= (0x9018 + 0) \mid 0 \\
 &= 0x9018
 \end{aligned}$$

Therefore, the dynamic linker must write the value `0x9018` to address `0x8024` to ensure that the function `foo` can access the variable `x`. It also needs to write the same value to address `0x9014` once the shared library is loaded into memory.

7.5 Handling function calls between modules

Because the execution addresses of different modules are not normally known until execution time, any function calls between the modules must be handled by the dynamic linker. To allow this the static linker generates relocations to be processed by the dynamic linker. The static linker might also insert code sequences known as PLT *entries* for the function calls (code generation for intra-call veneers). Whether the PLT entries are required is defined by the particular platform or operating system and is normally controlled with a linker command-line option.

Without PLT entries (using the `--pltgot=none` linker command-line option) the job of the static linker is not as complicated. However, the dynamic linker is restricted to loading the shared library to within the branch range of any application that wishes to call functions within that shared library. Also, the dynamic linker must resolve dynamic relocations for each instance of a function call. For example, if an application calls a shared library function `bar` eight times, the dynamic linker must resolve eight dynamic relocations.

PLT entries are commonly used because they can simplify the dynamic linking process. For an application that uses a function like `bar` from [Example 7-1 on page 7-6](#), the dynamic linker only needs to resolve a single relocation for its PLT entry. A PLT entry is a small piece of code inserted by the linker that can branch to anywhere in the 32-bit address range of the ARM CPU. It is effectively a long-branch veneer inserted between the caller and the dynamically imported callee or destination.

A PLT entry is made up of code inserted by the linker, a data entry, and a relocation. The data entry contains the address to branch to, and the relocation is against this location to allow the dynamic linker to enter the correct address. This means the dynamic linker only needs to insert the address of the target into the data entry and does not need to be concerned about calculating PC offsets or handling when the target function is out of direct branch range.

There are a number of different types of PLT entries. The instruction sequence generated and also where the data item is stored varies between the PLT types. Two of the most common are *direct* and *indirect* PLTs.

7.5.1 Direct PLTs

A direct PLT has the data entry stored with the PLT sequence code. Therefore a direct PLT looks like a veneer added to a statically linked image, apart from the relocation in the data entry. Therefore the code and data items are stored in the code section of the module. [Figure 7-1 on page 7-10](#) shows a direct PLT and the relocation against it.

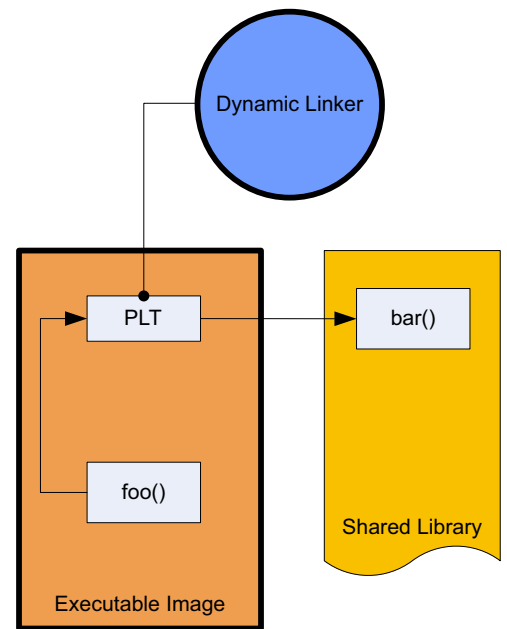


Figure 7-1 Function calls via direct PLTs

Example 7-3 Sample output from fromelf -c image

```

** Section #1 'ER_R0' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
   Size   : 40 bytes (alignment 4)
   Address: 0x00008000

   $t
   0x00008000:  4778      xG    BX    pc
   0x00008002:  46C0      .F    MOV    r8,r8

   $a
   0x00008004:  E59FC000   ....   LDR    r12,[pc,#0] ; [0x800C] = 0
   0x00008008:  E12FFF1C   ../.   BX     r12

   $d
   0x0000800C:  00000000   ....   DCD    0

   $a
   .text
   foo
   0x00008010:  E92D4010   .@-.   PUSH   {r4,lr}
   0x00008014:  EBFFFFFFA  ....   BL     0x8004 ; 0x8004
   0x00008018:  E59F0004   ....   LDR    r0,[pc,#4] ; [0x8024] = 0
   0x0000801C:  E5900000   ....   LDR    r0,[r0,#0]
   0x00008020:  E8BD8010   ....   POP    {r4,pc}

   $d
   0x00008024:  00000000   ....   DCD    0

```

This example output from fromelf shows the same code sequence for function foo, as described in [BPABI relocation example on page 7-6](#). However, this time the output shows the complete output for the section (#1), which also includes the PLT sequence from address 0x8000 to 0x800C. The BL instruction within function foo branches to the PLT sequence. The PLT sequence at address 0x8004 loads an address into register R12 and then branches to that address. The value loaded into R12 is located at address 0x800C, which is the first entry in the dynamic relocation section, as described in [BPABI relocation example on page 7-6](#). The dynamic linker must fill in the relocation when it loads the shared library into memory. This PLT sequence can be shared

across different units, so that other functions that call `bar` can do so dynamically. The final part of the PLT sequence, not already described, is the Thumb code from address `0x8000-0x8004`. This code, generated by the static linker, acts as a dynamic interworking veneer, so that it is possible for Thumb functions to safely call `bar`.

7.5.2 Indirect PLTs

An indirect PLT stores the data entry at a different location to the PLT code. The PLT code is placed in the code section of the executable. The data entry is normally stored in the data section of the executable, in a special area of the *Global Offset Table* (GOT) called the PLTGOT. The separation of the data entry and the code obviously means the code generated for an indirect PLT is different to that of a direct PLT as well as providing more flexibility. Rather than the dynamic linker writing to the PLT directly in the executable image, it writes the address of `bar` into the PLTGOT. The dynamic linker typically loads the PLTGOT into a more easily accessible read/write area of memory.

The target platform or operating system determines whether the GOT is generated by the dynamic linker or the static linker. [Figure 7-2](#) shows the GOT being generated by the dynamic linker, as in the BPABI model, whereas the GOT is generated by the static linker when targeting the SysV ARM Linux model.

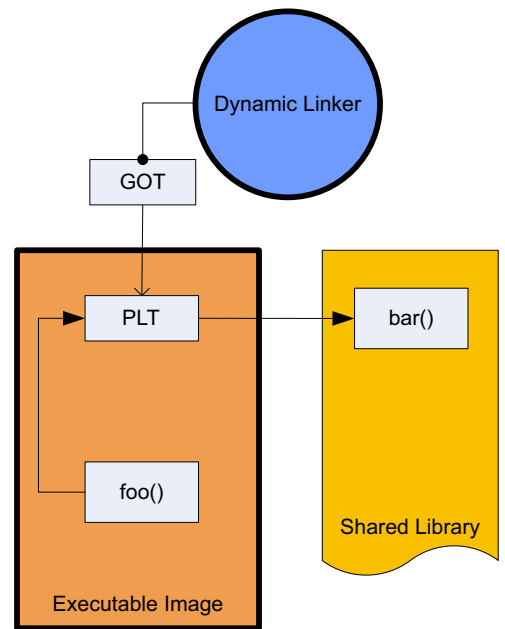


Figure 7-2 Function calls via indirect PLTs

The following example disassembly shows an indirect PLT and relocation.

```

** Section #1 'ER_R0' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
   Size   : 44 bytes (alignment 4)
   Address: 0x00008000

   $t
       0x00008000:  4778      xG    BX    pc
       0x00008002:  46C0      .F    MOV    r8,r8
   $a
       0x00008004:  E59FC004    ....   LDR    r12,[pc,#4] ; [0x8010] = 0
       0x00008008:  E59CC000    ....   LDR    r12,[r12,#0]
       0x0000800C:  E12FFF1C    ../.   BX     r12
   $d
       0x00008010:  00000000    ....   DCD    0
   $a
   .text
   foo
       0x00008014:  E92D4010    .@-.   PUSH   {r4,lr}
       0x00008018:  EBFFFFF9    ....   BL      0x8004 ; 0x8004
       0x0000801C:  E59F0004    ....   LDR    r0,[pc,#4] ; [0x8028] = 0
       0x00008020:  E5900000    ....   LDR    r0,[r0,#0]
       0x00008024:  E8BD8010    ....   POP     {r4,pc}
   $d
       0x00008028:  00000000    ....   DCD    0

:
:

** Section #5 '.dyn' (SHT_REL)
   Size   : 16 bytes (alignment 4)
   Symbol table #3 '.dynsym'
   2 relocations applied to section #0 '[Anonymous Section]'

#  Offset      Relocation Type  Wrt  Symbol  Defined in
0  0x00008010   95 R_ARM_GOT_ABS    2    bar    Ref
1  0x00008028    2 R_ARM_ABS32      3     x     Ref

```

There are two obvious differences:

- a new type of relocation is present in the dynamic relocation section called `R_ARM_GOT_ABS`.
- two LDR instructions are generated adding an extra level of redirection.

The `R_ARM_GOT_ABS` relocation instructs the dynamic linker to provide the absolute address of the GOT entry. The first LDR instruction loads into register R12 the address of the GOT entry, determined by the dynamic linker.

7.6 Handling data accesses between modules

Accessing data between different modules is a similar problem to resolving function calls between modules, because at static link time the addresses of the data are unknown to the static linker and must be handled by the dynamic linker.

When accessing a data item from another module, the object producer must add an extra level of indirection to the data access. This allows the use of a table of addresses to store the actual address to be accessed, this table is known as the *Global Offset Table* (GOT).

In a statically linked image on an ARM processor, one level of indirection is used, for example:

```

        LDR    r0, [X]
        LDR    r0, [r0]
X   DCD    &Data

```

In this example, the address where the data is stored is known at static link time and is stored at the literal pool labeled X. For a dynamically linked image, the address where the data item is stored is not known, but the location in the GOT is known. Therefore the object producer must add an extra level of indirection. For example:

```

        LDR    r0, [X]
        LDR    r0, [r0]
        LDR    r0, [r0]
X   DCD    &GOT_Entry

```

This code generates the following sequence of operations:

1. The first instruction loads the contents of the literal pool. That is, the GOT location.
2. The second instruction loads the address from the GOT.
3. The third instruction loads the actual data.

Chapter 8

Controlling what symbols can be dynamically linked

This section includes:

- *Symbol visibility* on page 8-2
- *Populating the dynamic symbol table* on page 8-4
- *Symbol preemption* on page 8-5
- *Symbol versioning* on page 8-8.

8.1 Symbol visibility

Within a module there might be functions and data that must be made available to other modules. Conversely there might be functions and data that are private to a module.

To control whether symbol definitions are made available, each symbol has an associated visibility property. The symbol visibility of a definition controls whether a symbol is visible externally to the module and also whether the symbol can be pre-empted at run-time.

The symbol visibility also applies to a symbol reference. Therefore, symbol visibility can be used to control whether particular references can be resolved at dynamic link time or must be resolved as static link time.

ELF supports four different visibility properties, default, protected, hidden and internal. [Table 8-1](#) summarizes these:

Table 8-1 ELF visibility

Visibility	ELF Name	Dynamically linkable?
Default	STV_DEFAULT	Yes
Protected	STV_PROTECTED	Yes
Hidden	STV_HIDDEN	No
Internal	STV_INTERNAL	No

In addition to visibility, ELF also supports a binding property. This property is separate to the visibility property and is normally used to indicate, *global*, *local*, or *weak binding* for a symbol. The symbol binding property is normally used by the object producer to control how the static linker resolves references to that symbol. For example, static symbols in C normally have local binding which prevents the static linker resolving references from other object files to the local definition.

8.1.1 Setting symbol visibility

The visibility of a symbol or reference can be set in a number of ways. It is normally set by the object producer, but most static linkers also provide a method to override the visibility of particular symbols.

An object producer normally has a default visibility setting for symbols, not to be confused with `STV_DEFAULT` here. For the ARM Compiler toolchain this is `STV_HIDDEN`. However, the object producer normally provides methods to override this either globally or for individual symbols. Globally overriding the visibility is normally done using a command-line option, for example `--no_hide_all` with `armcc`.

Overriding individual symbols is normally achieved using a source code annotation. Therefore users must individually mark the symbols they want to change the visibility of. There are a number of different annotations used to do this. The two most common annotations are:

- GNU visibility attributes: `__attribute__((visibility("xxx")))`
- Microsoft-style declaration specifications: `__declspec(attributes)`.¹

The ARM compiler (`armcc`) supports both of these annotations.

1. While it is true that they use Microsoft syntax, they are used in a different way from the way they are used in Microsoft's COFF toolchain. When building a COFF DLL, you make that DLL's header file conditional by using `__declspec(dllexport)` on all declarations instead of `__declspec(dllimport)`. When building a Symbian DLL with the ARM Compiler toolchain, you normally mark the declarations in DLLs header file with the `__declspec(dllimport)` attribute and then override the attribute with `__declspec(dllexport)` for the definitions in the corresponding source file.

8.2 Populating the dynamic symbol table

As mentioned in [Chapter 6 ELF overview and background](#), an executable module might have two symbol tables, a static symbol table and dynamic symbol table. The static symbol table is generated by the static linker from the symbol tables of the input object files. It is not used by a dynamic linker and is normally removed from the ELF in release modules.

The dynamic symbol table is used by the dynamic linker to identify the symbols defined and referenced in a module. The dynamic symbol table is also created by the static linker. How the static linker populates the dynamic symbol table is toolchain and platform specific.

Static linkers normally use a fixed method, or alternatively allow the user to select how to populate the dynamic symbol table, giving them full control. The latter approach requires the user to specify which symbols must be present in the dynamic symbol table. This is normally achieved using a textual input file to the linker.

Another method, probably the most common, is to populate the dynamic symbol table based on the symbol visibility in the static symbol table. Symbols with `STV_DEFAULT` or `STV_PROTECTED` visibility are automatically added to the dynamic symbol table. Symbols with `STV_HIDDEN` or `STV_INTERNAL` visibility are not normally added to the dynamic symbol table.

8.3 Symbol preemption

Some platforms allow symbols to be preempted at run-time. This means that the dynamic loader can load a number of implementations of a particular function and select the most appropriate implementation at run-time. This is normally achieved by using version information embedded into the executable module.

The selected implementation is used for all STV_DEFAULT visibility references to that particular symbol, even when the module itself contains a definition of the symbol being preempted.

For a particular symbol to be preempted it must have STV_DEFAULT visibility. Symbols given STV_PROTECTED visibility cannot be preempted at run-time.

A preemptable symbol definition can be preempted by the dynamic linker. This means that the dynamic linker might use a different definition of the symbol at run-time. Therefore all references to the symbol must be made as if the symbol is external to the module (given STV_DEFAULT visibility) and the static linker leaves it to the dynamic linker to handle the symbol reference at run-time. This means that the static linker creates a relocation and possibly a PLT entry for each function call. This allows the dynamic linker to retarget the branch instruction or PLT sequence to the correct pre-empting function. PLTs are described in more detail in [Handling function calls between modules on page 7-9](#).

STV_PROTECTED symbols cannot be preempted at run-time. Any function calls from the same module to an STV_PROTECTED symbol are not directed using a PLT. A branch instruction is generated instead. However, the PLT is still generated by the static linker because it does not know whether other modules make use of such functions.

8.3.1 SysV symbol preemption example

Example 8-1 shared.c

```
int x = 1;

void multiply(void)
{
    x = x * x;
}

__attribute__((visibility("protected"))) void increment(void)
{
    x++;
}

void bar(void)
{
    increment();
    multiply();
}
```

In [Example 8-1](#) there are three functions in the shared object: `bar`, `multiply` and `increment`. Both `bar` and `multiply` are given STV_DEFAULT visibility (the default for SysV). However, `increment` has been given STV_PROTECTED visibility.

Example 8-2 Sample output from fromelf -csry shared.so

```

** Section #10 '.plt' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
Size   : 64 bytes (alignment 4)
Address: 0x000003B4

$a
0x000003B4: E52DE004    ..-.  PUSH    {lr}
0x000003B8: E28FEC82    ....  ADR     lr,{pc}+0x8208 ; 0x85C0
0x000003BC: E28EE078    x...  ADD     lr,lr,#0x78
0x000003C0: E5BEF000    ....  LDR     pc,[lr,#0]!

$t
0x000003C4: 4778        xG     BX     pc
0x000003C6: 46C0        .F     MOV     r8,r8

$a
[Anonymous symbol #31]
0x000003C8: E28FCC82    ....  ADR     r12,{pc}+0x8208 ; 0x85D0
0x000003CC: E28CC06C    l...  ADD     r12,r12,#0x6C
0x000003D0: E5BCF000    ....  LDR     pc,[r12,#0]!

$t
0x000003D4: 4778        xG     BX     pc
0x000003D6: 46C0        .F     MOV     r8,r8

$a
[Anonymous symbol #32]
0x000003D8: E28FCC82    ....  ADR     r12,{pc}+0x8208 ; 0x85E0
0x000003DC: E28CC060    `...  ADD     r12,r12,#0x60
0x000003E0: E5BCF000    ....  LDR     pc,[r12,#0]!

$t
0x000003E4: 4778        xG     BX     pc
0x000003E6: 46C0        .F     MOV     r8,r8

$a
[Anonymous symbol #33]
0x000003E8: E28FCC82    ....  ADR     r12,{pc}+0x8208 ; 0x85F0
0x000003EC: E28CC054    T...  ADD     r12,r12,#0x54
0x000003F0: E5BCF000    ....  LDR     pc,[r12,#0]!

** Section #11 '.text' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
Size   : 272 bytes (alignment 4)
Address: 0x000003F4

$a
[Anonymous symbol #23]
multiply
0x000003F4: E59F1038    8...  LDR     r1,[pc,#56] ; [0x434] = 0x8248
0x000003F8: E79F1001    ....  LDR     r1,[pc,r1]
0x000003FC: E5910000    ....  LDR     r0,[r1,#0]
0x00000400: E0020090    ....  MUL     r2,r0,r0
0x00000404: E5812000    . ..  STR     r2,[r1,#0]
0x00000408: E12FFF1E    ../.  BX      lr

increment
0x0000040C: E59F0024    $....  LDR     r0,[pc,#36] ; [0x438] = 0x8230
0x00000410: E79F0000    ....  LDR     r0,[pc,r0]
0x00000414: E5901000    ....  LDR     r1,[r0,#0]
0x00000418: E2811001    ....  ADD     r1,r1,#1
0x0000041C: E5801000    ....  STR     r1,[r0,#0]
0x00000420: E12FFF1E    ../.  BX      lr

bar
0x00000424: E92D4010    .@-.  PUSH    {r4,lr}
0x00000428: EBFFFFFF7    ....  BL      increment ; 0x40C
0x0000042C: EBFFFFFF5    ....  BL      0x3C8 ; 0x3C8
0x00000430: E8BD8010    ....  POP     {r4,pc}

$d

```

```

0x00000434: 00008248 H... DCD 33352
0x00000438: 00008230 0... DCD 33328
:
:
:
** Section #19 '.got' (SHT_PROGBITS) [SHF_ALLOC + SHF_WRITE]
Size : 40 bytes (alignment 4)
Address: 0x00008630

0x008630: 48 85 00 00 00 00 00 00 00 00 00 00 B4 03 00 00
0x008640: B4 03 00 00 B4 03 00 00 00 00 00 00 00 00 00 00
0x008650: 00 00 00 00 00 00 00 00

```

There are two function calls in the bar function. The first call is to the increment function using a standard branch with link (BL) instruction. This is because the increment symbol cannot be preempted. The second call to the multiply function, which can be preempted, uses a three step indirect PLT sequence.

-
1. Branch with link to address 0x3C8 of the .plt section (the PLT entry for the multiply function).
 2. Load into register R12 address 0x863C from the .got section (0x85D0 + 0x6C), which is the address of the PLTGOT entry for the multiply function.
 3. Load into the program counter the address of the multiply function from the PLTGOT data entry at address 0x863C.¹.

The call to a non-preemptive, STV_PROTECTED symbol such as increment has the advantage of being faster because only a single branch instruction is required. It also means that the reference can be resolved at static link time, which also speeds-up and simplifies the dynamic linking step. However, unlike the call to a non-preemptive function, a call to a preemptive function always uses a PLT entry.

Because [Example 8-1 on page 8-5](#) is targeted for ARM Linux, the PLT entries are also indirect. Also, note that unlike the example in [Indirect PLTs on page 7-11](#), where no GOT is generated by the static linker because the BPABI linking model is used, a .got section is generated by the static linker when targeting the SysV ARM Linux model.

Note

Preemption is related to function calls only. Data accesses to symbols given STV_PROTECTED visibility are reached using a GOT entry.

Note

In the BPABI model, symbol preemption is optional. See the BPABI for the ARM Architecture document for details.

1. This address must be resolved by the dynamic linker, which in [Example 8-1 on page 8-5](#) is the ARM Linux kernel.

8.4 Symbol versioning

Symbol versioning records extra information about a particular symbol imported from or exported from a shared library. A shared library can be updated over time. For example, a shared library might contain a function called `foo`. The original version of the shared library might be linked against an application which calls `foo`. In the future the shared library may be updated with a new version of function `foo`, which could contain an update. However, an old application may be tied to the original version of function `foo`, so both new and old versions of `foo` must be available in the shared library. Unfortunately, it is not possible to build a source file with two functions with exactly the same name because the static linker generates a multiply defined symbol error message. Symbol versioning works around this problem.

The `examples/example_8-4` directory of this application note contains three build scripts. The first build script links an application against the original shared library, which contains the original function `foo`. The second build script links an application against a second version of the shared library, which contains the original version of `foo` and a new default version of `foo`. Finally, the third build script links an application against a third version of the shared library, which contains three versions of `foo`: the original version, the second version, and a new default version.

Where there are multiple versions of a given symbol like `foo`, by default `armlink` selects the default version of that symbol. In symbol versioning the default symbol is denoted by the `@@` characters, for example:

```
int foo1(void) __asm__("foo@ver1");
int foo2(void) __asm__("foo@ver2");
int foo3(void) __asm__("foo@@ver3");
```

In this example from `shared_v3.c` in example 8-4, the third version of `foo` is the default version. If an application is linked against an older version of the symbol, the dynamic linker can detect this and fix the dynamic relocations against the correct version.

Finally, it is possible for an application to reference a symbol which is defined in multiple shared libraries. For example, the following diagram shows three modules (shared libraries), each containing the same version of the symbol `foo`.

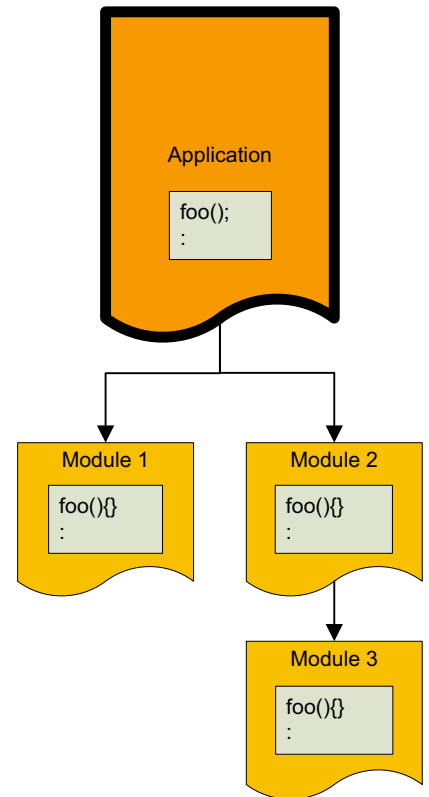


Figure 8-1 Choosing a definition of a multiply defined symbol

The dynamic linker typically performs a breadth-first search algorithm until it finds a suitable definition to resolve the reference.

In addition to the `__asm__` syntax used in example 8-4, it is also possible to write a *symbol version script*. For more information about the scripting language, see the linker documentation.

Chapter 9

Dynamic linking models

The ARM Compiler toolchain supports various type of images including bare-metal and dynamically linked images. This section looks at the different types of images and their associated build options. It includes:

- *Bare-metal* on page 9-2
- *DLL-like* on page 9-3
- *SysV* on page 9-5
- *Choosing an appropriate model* on page 9-7.

9.1 Bare-metal

The *bare-metal* linking model only requires a static linker. No dynamic linking is necessary. A single address space is used and there is no virtual address space, because the virtual address of a symbol equals its physical address.

Typically, the memory layout is determined at static link time, although a bare-metal system might include features such as overlay management, as described in [Chapter 5 Alternatives to dynamic linking](#). Although a bare metal system does not involve any complex operating system or dynamically loaded executables, it might contain a *Real-time Operating System* (RTOS) which is statically linked with the executable in ROM.

Examples 3-1 and 3-2 are simple bare-metal applications.

9.2 DLL-like

This section includes:

- [Base Platform Application Binary Interface \(BPABI\) linking model](#)
- [The base platform linking model.](#)

9.2.1 Base Platform Application Binary Interface (BPABI) linking model

The BPABI memory model in scatter-loading format is:

```
LR_1 <read-only base address> ; default 0x8000
{
    ER_R0  +0
    {
        *(+R0)
    }
}

LR_2 <read-write base address> ; default 0x0
{
    ER_RW  +0
    {
        *(+RW)
    }
    ER_ZI  +0
    {
        *(+ZI)
    }
}
```

This works well for loadable applications or DLLs. In most cases, you must specify the `--ro_base` and `--rw_base` options, because the default values, `0x8000` and `0x0` respectively, might not be suitable for your platform. These addresses do not have to reflect the addresses to which the image is relocated at run time.

The output from example 7-1 confirms that the default read-only base address of `0x8000` is used when building for the BPABI linking model. In fact, the default memory layout for the BPABI model is the same as the bare-metal linking model.

Most compiler options are supported under the BPABI model. The options specific to the BPABI model are linker options. To build a BPABI executable, the `--bpabi` option must be specified. When building a shared library, then the `--d11` option must also be specified to `armlink`. The final option to consider is the `--pltgot=<type>` option, where `<type>` equals `none`, `direct`, `indirect` or `sbrel`. For more information, see [Other dynamic image related linker \(armlink\) options on page 10-5](#), and also the linker documentation

9.2.2 The base platform linking model

The BPABI model described in [Base Platform Application Binary Interface \(BPABI\) linking model](#) has a fixed memory map. Unfortunately, this prevents users of a non-contiguous memory model from using the BPABI model. In the past, it was possible to use the legacy model (`--reloc`), however, this model is only partially supported and documented, and moreover uses non ABI-compliant dynamic relocations. Therefore, the BPABI model is not suitable for bare-metal systems that have a fixed static address but need to access some dynamic content.

In RVCT 4.0 and later a new `--base_platform` option exists, which merges the features of the BPABI and legacy models. This means that users of a non-contiguous memory model can link dynamically and also describe their memory layout using a scatter-loading description file. Not only is it possible to describe the memory map of the executable image, but it is also possible to

describe the memory map of the shared library. Being able to describe the memory layout allows the user to give extra information to the dynamic linker. The following example scatter-loading description file below shows a critical code section being placed at address 0x9000 (fast memory), and a non-critical code section being placed at address 0x10000 (slow memory). This scatter-file describes the ideal memory layout for different sections within a shared library. The RELOC scatter-loading attribute informs armlink that the sections are relocatable and that the linker must generate dynamic relocations for any module trying to access these sections.

```
LR1 0x9000 RELOC
{
    FAST_MEMORY +0
    {
        ; place critical function here
    }
}

LR2 0x10000 RELOC
{
    SLOW_MEMORY +0
    {
        ; place non-critical function here
    }
}
```

The dynamic linker can use this extra information to place each section appropriately. If another module already exists at the address described in the scatter-file, and it is undesirable to overwrite that module, the dynamic linker loads the relevant section to the next most suitable address. If there is no more memory available at all, then the dynamic linker has to unload or overwrite an existing, low priority module, and reload it again later if required.

The examples/example_9-2 directory, provided as part of this application note, shows how to build a shared library targeted at the Base Platform linking model. Additionally, many more example scatter-files for the Base Platform model are available in:

ARM Compiler toolchain Using the Linker,
<http://infocenter.arm.com/help/topic/com.arm.doc.dui0474-/CHDDDBGE.html>.

9.3 SysV

SystemV Release 4, commonly written as SysV or SysVr4 is the executable format commonly used on UNIX type operating systems. The images are stored and executed directly from the ELF file format.

The ARM Compiler toolchain can be used to build both SysVr4 executables and shared libraries. It also supports the creation of ARM Linux executables and shared libraries. Details on this can be found in separate application notes and product documentation.

A small number of build options are required to build SysV executables. Building SysV shared libraries requires some additional options.

For the compiler, `--no_hide_all` is normally used. As previously mentioned, in a SysV image the intention is that it is transparent to the programmer whether a symbol is statically or dynamically linked. Compiling with the `--no_hide_all` option gives all symbols STV_DEFAULT visibility, which allows them to be dynamically or statically linked.

9.3.1 Building SysV executables

To link a SysV executable `--sysv` must be used on the linker command line. Also a subset of the GNU ld script language was added in ARM Compiler toolchain 4.1 to give finer control over the layout of sections in the SysV output of the linker.

It is possible to build a SysV executable with position independence by compiling with the `--apcs=/fpic` option. This is not required by some SysV systems, for example, ARM Linux executables always execute from a fixed address of `0x8000`. However, other operating systems based on the SysV model might decide to have position independent executables.

Executables being statically linked at fixed addresses have an interesting side effect when they try to access variables from shared libraries. That is, if the application is built without position independence, the extra level of indirection required to access the variables in a dynamic linked system is not generated. Therefore the variable must be accessed within the executables address space. However, because the variable is in a shared library it is not in the address space of the executable. To overcome this problem the linker allocates space in the data section of the executable and generates a *copy relocation*.

The copy relocation instructs the dynamic linker to copy the value of the variable from the shared library to the location in the application. References from the shared library to the variable are re-targeted to the copy in the executable.

9.3.2 Building shared objects

To build SysV shared libraries the code must be compiled for position independence using the `--apcs=/fpic` option. This is because a shared library can be loaded to any suitable address in the memory map. Again, the compiler option `--no_hide_all` is normally used. The linker options required to build a SysV shared library are `--sysv` and `--shared`.

9.3.3 Symbol visibility rules

The linker has a set of rules for which symbols are to be added to the to the dynamic symbol table. [Table 9-1 on page 9-6](#) summarizes these rules.

Table 9-1 Visibility rules for references and definitions

Visibility	Reference or Definition	Executable	Shared Library
STV_DEFAULT	Reference	Yes	Yes
STV_DEFAULT	Definition	No	Yes
STV_PROTECTED	Reference	Yes	Yes
STV_PROTECTED	Definition	No	Yes
STV_HIDDEN	Reference	No	No ^a
STV_HIDDEN	Definition	No	No ^a
STV_INTERNAL	Reference	No	No ^a
STV_INTERNAL	Definition	No	No ^a

- a. STV_HIDDEN or STV_INTERNAL global symbols that are required for relocation can be placed in the dynamic symbol table, but the linker changes them into local symbols to prevent them from being accessed from outside the shared library.

9.4 Choosing an appropriate model

It is important for the ARM linker (arm1ink) to understand the relationship between the toolchain and the platform. The BPABI is a base standard to which arm1ink complies and means that further processing might be required.

As mentioned previously the split of responsibility between static and dynamic linking can be made at many different points. For a platform holder in charge of the toolchain the two can be designed symbiotically. For arm1ink, the owner of the toolchain (ARM) is different to the owner of the platform. Therefore arm1ink must provide several models, each encoding a different split of responsibility between the static and dynamic linker. The designer of the platform must choose the model that most closely matches the design goals of the platform.

9.4.1 Single address space versus multiple address spaces

A system requires a single address space or multiple address spaces, and is a key factor for deciding if it is necessary to use a dynamic linker. For example, if a system does not have a *Memory Management Unit* (MMU), and requires loadable modules only, then it does not need to consider the SysV model because it requires only a single address space for each process.

9.4.2 SysV versus BPABI

The SysV model is compatible with Linux applications and DSOs. The BPABI model is compatible with Symbian and Microsoft Windows-like applications and DLLs. The BPABI model might require post linker support, whereas SysV executables and DSOs do not require any further processing because they can be handled by an ARM Linux kernel.

9.4.3 BPABI versus base platform

The BPABI is a standard defined by the BPABI for the ARM Architecture. Other compilers such as GCC can target the BPABI. Therefore a platform owner providing a post-linker does not get locked into using the ARM Compiler.

The Base Platform is an extension to the BPABI which is most useful in producing a platform with a single address space application, but with loadable module support. When an MMU exists there is little need for the custom memory map that it provides.

Chapter 10

Dynamic linking with the ARM Compiler toolchain

This section describes the features available in the ARM Compiler toolchain for generating dynamic images. This includes different build options and language extensions.

The various tools features and switches are covered in this section but only a limited amount of information is provided. See the ARM Compiler toolchain documentation for more information.

This section includes:

- [Controlling symbol visibility in the ARM Compiler toolchain on page 10-2](#)
- [Controlling dynamic section contents on page 10-7](#).

10.1 Controlling symbol visibility in the ARM Compiler toolchain

This section includes:

- [Compiler \(armcc\)](#)
- [Assembler \(armasm\)](#) on page 10-3
- [Linker \(armlink\)](#) on page 10-3
- [Steering files](#) on page 10-4
- [Dynamic symbol table rules](#) on page 10-4
- [ELF reader and converter \(fromelf\)](#) on page 10-5
- [Other dynamic image related linker \(armlink\) options](#) on page 10-5.

10.1.1 Compiler (armcc)

The ARM compiler (armcc) supports two basic methods to control symbol visibility, command-line switches and source code annotation. [Table 10-1](#) summarizes the compiler command-line options.

Table 10-1 Dynamic linking compiler options

Option	Supported from	Description
--[no_]hide_all	RVCT 2.2	Gives all symbol definitions STV_DEFAULT visibility
--[no_]dllexport_all	RVCT 2.2	Marks all extern definitions as: __declspec(dllexport)
--[no_]export_defs_implicitly	RVCT 2.2	Exports all functions marked as: __declspec(dllimport)
--[no_]export_all_vtbl	RVCT 2.2	Exports all virtual table functions and <i>RunTime Type Information</i> (RTTI)
--[no_]dllimport_runtime	RVCT 3.1	Marks run-time functions, e.g. C library functions, as: __declspec(dllimport)

Also, armcc supports a number of source code annotations to control visibility. The behaviour of some of these has changed between releases of the ARM compiler. [Table 10-2](#) shows the annotations and behaviour in releases.

Table 10-2 Dynamic object compiler options

Option	Toolchain	Description
__declspec(dllexport)	RVCT 2.2	Gives symbol definitions STV_DEFAULT visibility, gives symbol references STV_HIDDEN visibility, and creates a linker EXPORT directive ^a
__declspec(dllimport)	RVCT 2.2	Gives symbol references STV_DEFAULT visibility and creates a linker IMPORT directive ^a
__declspec(dllexport)	RVCT 3.0, 3.1	Gives symbol definitions STV_DEFAULT visibility and gives symbol references STV_HIDDEN visibility
__declspec(dllimport)	RVCT 3.0, 3.1	Gives symbol references STV_DEFAULT visibility
__declspec(dllexport)	RVCT 4.0, ARM Compiler toolchain 4.1	Gives symbol definitions and references STV_PROTECTED visibility.
__declspec(dllimport)	RVCT 4.0, ARM Compiler toolchain 4.1	Gives symbol references STV_PROTECTED visibility

Table 10-2 Dynamic object compiler options (continued)

Option	Toolchain	Description
<code>__attribute__((visibility("default")))</code>	RVCT 4.0, ARM Compiler toolchain 4.1	Gives the symbol definition or reference STV_DEFAULT visibility
<code>__attribute__((visibility("protected")))</code>	RVCT 4.0, ARM Compiler toolchain 4.1	Gives the symbol definition or reference STV_PROTECTED visibility
<code>__attribute__((visibility("hidden")))</code>	RVCT 4.0, ARM Compiler toolchain 4.1	Gives the symbol definition or reference STV_HIDDEN visibility
<code>__attribute__((visibility("internal")))</code>	RVCT 4.0, ARM Compiler toolchain 4.1	Gives the symbol definition or reference STV_INTERNAL visibility

- a. The RVCT 2.2 compiler also generates `.directive` sections. These sections embed steering file commands into an ELF object file for any symbol whose destination address was unknown. These must be processed by the linker. This requirement has been superseded by ELF-standard mechanisms. Therefore, in RVCT 3.0 and later, these sections are no longer generated by the compiler. The annotations in [Table 10-3](#) and steering files can be used instead. For more details about steering files, see [Steering files on page 10-4](#).

Further information on symbol visibility is available in the ARM Compiler toolchain Migration and Compatibility document.

10.1.2 Assembler (armasm)

Like the compiler the symbol visibility can be controlled by assembler command-line options or in the source code. Also, the assembler gives symbol definitions and references hidden visibility by default. You can change this to STV_DEFAULT visibility using the `--no_hide_all` option.

The symbol visibility can also be controlled in the source code. In the assembler, symbols remain local unless they are imported, using the `IMPORT` directive, or exported with the `EXPORT` directive. You can add an extra attribute to control the visibility.

Table 10-3 Dynamic object assembler directives

Option	Supported from	Visibility
<code>IMPORT foo</code> <code>EXPORT bar</code>	RVCT 2.2	STV_HIDDEN
<code>IMPORT foo,DYNAMIC</code> <code>EXPORT bar,DYNAMIC</code>	RVCT 2.2	STV_DEFAULT
<code>IMPORT foo,PROTECTED</code> <code>EXPORT bar,PROTECTED</code>	RVCT 2.2	STV_PROTECTED
<code>IMPORT foo,HIDDEN</code> <code>EXPORT bar,HIDDEN</code>	RVCT 2.2	STV_HIDDEN
<code>IMPORT foo,INTERNAL</code> <code>EXPORT bar,INTERNAL</code>	RVCT 3.1	STV_INTERNAL

10.1.3 Linker (armlink)

Symbol visibility is particularly important in the linker. This is where the visibility of symbol definitions and references are checked to ensure that they are suitable. For example, the linker ensures a hidden visibility symbol is not attempting to be used dynamically. There is an `armlink`

diagnostic option `--info visibility` which when used with `--verbose` shows the transitions in visibility that a symbol goes through. The linker is also responsible for populating the dynamic symbol table and creating the `.dynamic` ELF section.

The ELF specification provides some rules on how the linker must handle symbol visibility when there is a difference between symbol definition and reference visibility. The ELF specification says that the linker must use the most restrictive visibility of any of the symbol definitions or references. The ELF specification classes `STV_INTERNAL` as the most restrictive and `STV_DEFAULT` as the least restrictive.

In practice this means an `STV_HIDDEN` visibility reference to an `STV_DEFAULT` visibility definition results in an `STV_HIDDEN` visibility definition. `arm1ink` allows this to be overridden by specifically importing or exporting a symbol in a steering file or by selecting to always use the definition visibility for the output visibility.

10.1.4 Steering files

Steering files can be used to change the visibility of a symbol from `STV_HIDDEN` visibility to `STV_DEFAULT` visibility. A steering file is a textual file containing commands for the linker that is supplied with the `--edit <steering_file>` command-line option. There are two steering file commands:

- `IMPORT` controls symbol references
- `EXPORT` controls symbol definitions.

`IMPORT` changes the visibility of references from `STV_HIDDEN` to `STV_DEFAULT`. `EXPORT` changes the visibility of definitions from `STV_HIDDEN` to `STV_DEFAULT`. Both commands also cause the symbol reference or definition to be added to the dynamic symbol table. Therefore these steering file commands can be used to manually control the contents of the dynamic symbol table.

———— Note ————

In RVCT 4.0 and later the `--override_visibility` linker switch is required for the `EXPORT` and `IMPORT` steering file commands to change the visibility of symbols.

10.1.5 Dynamic symbol table rules

As well as manual control of the contents of the dynamic symbol table, `arm1ink` also supports populating the dynamic symbol table using a number of fixed exporting rules. These fixed exporting rules use the symbol visibility and type information from the static symbol table to decide which symbols are to be added to the dynamic symbol table.

The exact rules used by the linker depend on the linker command-line options and the type of image being generated by the linker. There are still a number of linker switches that control the exporting rules used to populate the dynamic symbol table. These are listed in [Table 10-4](#).

Table 10-4 Dynamic image linker options

Option	Supported from	Visibility
<code>--bpabi</code>	RVCT 2.2	<code>STV_HIDDEN</code>
<code>--dll {--bpabi}</code>	RVCT 2.2	<code>STV_DEFAULT</code>
<code>--max_visibility=protected</code>	RVCT 2.2	<code>STV_HIDDEN</code>
<code>--[no_]export_all</code>	RVCT 2.2	<code>STV_PROTECTED</code>

Table 10-4 Dynamic image linker options (continued)

Option	Supported from	Visibility
--shared {--sysv}	RVCT 2.2	Creates a SysV Shared library and select SysV shared linking exporting rules
--sysv	RVCT 2.2	Creates a SysV executable and selects SysV shared linking exporting rules
--override_visibility	RVCT 3.1	STV_INTERNAL
--base_platform	RVCT 4.0	STV_DEFAULT
--undefined=symbol	ARM Compiler toolchain 4.1	Create a symbol reference to the specified symbol name. Issue an implicit --keep(symbol) to prevent any sections brought in to define that symbol from being removed.
--undefined_and_export=symbol	ARM Compiler toolchain 4.1	The same behaviour as the --undefined=symbol, but also adds an implicit EXPORT symbol to push the specified symbol into the dynamic symbol table
--use_definition_visibility	ARM Compiler toolchain 4.1	A symbol reference with STV_HIDDEN visibility combined with a definition with STV_DEFAULT visibility results in a definition with STV_HIDDEN visibility.

10.1.6 ELF reader and converter (fromelf)

fromelf can be used to view the symbol visibility of symbols from different ELF files. This is shown in the Vis column of the symbol table output (fromelf -s <file>) generated from objects, static or shared libraries, and executable images.

In RVCT 4.0 and later fromelf can also be used to change the visibility of symbols:

- --show can be used to change the visibility of a symbol to STV_DEFAULT
- --hide can be used to change the visibility of a symbol to STV_HIDDEN.

10.1.7 Other dynamic image related linker (armlink) options

The linker switches shown in [Table 10-5](#) are related to the creation of dynamic images:

Table 10-5 Additional dynamic image linker options

Option	Supported from	Visibility
--cppinit=symbol	RVCT 2.2	Controls the C++ initialisation function used by the image
--fpic	RVCT 2.2	Causes the linker to generate a position independent image, code must have been compiled with --apcs=/fpic
--force_so_throw	RVCT 2.2	Force the linker to keep C++ exception unwinding tables even if C++ exceptions are not used
--linux_abitag=version_id	RVCT 2.2	Specify the minimum compatible Linux kernel version
--dynamic_debug	RVCT 3.0	Causes the linker to output relocations for the debug data. This can aid the debugging of dynamic images in debuggers support this functionality

Table 10-5 Additional dynamic image linker options (continued)

Option	Supported from	Visibility
<code>--arm_linux</code>	RVCT 4.0	Set default switches for building ARM Linux applications
<code>--library=name</code>	RVCT 4.0	Adds libname.a or libname.so to the linker command line based on the <code>--[no_]search_dynamic_libraries</code> option
<code>--pltgot=type</code>	RVCT 4.0	Generate tables corresponding to the different addressing modes of the BPABI

10.2 Controlling dynamic section contents

armlink provides a number of features to control and add information to the .dynamic section generated in a dynamic image. These are mostly implemented with linker command-line options shown in [Table 10-6](#):

Table 10-6 Controlling dynamic section contents with armlink

Option	Supported from	Visibility
--runpath=pathname	RVCT 3.1	Adds extra locations, DT_RPATH tags, to be search by the dynamic linker for dependencies
--dynamiclinker=name	RVCT 4.0	Control the dynamic linker set in the executable via the PT_INTERP program header element
--fini=symbol	RVCT 4.0	Set the finalization symbol, DT_FINI, for an image
--init=symbol	RVCT 4.0	Set the initialization symbol, DT_INIT, for an image
--no_add_needed	RVCT 4.0	Causes the linker to add dependencies of shared libraries to dependency list, DT_NEEDED tags, of current image
--prelink_support	RVCT 4.0	Causes changes in ELF to allow prelink use, includes generating extra DT_NULL tags to create space in the dynamic symbol table for prelink
--soname=name	RVCT 4.0	Set the name, DT_SONAME, of SysV shared library
--linker_script=ld_script	ARM Compiler toolchain 4.1	Implements a subset of the GNU ld script language to give finer control over the layout of sections in the SysV output of the linker

As well as these linker switches there is a single steering file command, REQUIRE, than can be used to make the linker add extra dependencies, DT_NEEDED tags, into the dynamic section.

Glossary

This glossary lists abbreviations used in this document.

armasm	The ARM Assembler.
armcc	The ARM C/C++ Compiler.
armlink	The ARM Linker.
ABI	Application Binary Interface.
ARM Compiler	The ARM Compiler is the only commercial compiler co-developed with the ARM processors and specifically designed to optimally support the ARM architecture.
ARM Compiler toolchain	After RVCT 4.0, RVCT became known as the ARM Compiler toolchain.
Assembler	A program that translates assembly level source code written in a given language, for example, ARM assembly language, into another language, for example, ELF.
BPABI	Base Platform Application Binary Interface.
Compiler	A program that translates source code written in a given language, for example, C or C++, into another language, for example, ELF.
DLL	See Dynamically Linked Library .
DSO	See Dynamically Shared Object .
Dynamic Linker	A post linking tool typically used to dynamically resolve unresolved references at run-time.
Dynamic Loader	A post linking tool typically used to dynamically resolve references at load time.
Dynamically Linked Library	A library, also known as a DLL or DSO, that can loaded at run-time.

Dynamically Shared Object	See Dynamically Linked Library .
ELF	Executable and Linking Format.
ELF Object	An intermediate ELF file that still contains unresolved references.
ELF Image	An ELF executable image that can be loaded and run on a target system either immediately or after additional processing.
fromelf	The ARM ELF Reader and Image Converter.
GOT	See Global Offset Table .
Global Offset Table	A table containing offsets to symbols that might be resolved at load or run-time.
GCC	The GNU Compiler Collection.
GNU	The GNU project, which recursively stands for "GNU not UNIX".
Image	An ELF image generated by armlink.
Linker	A binding tool that resolves symbol references to symbol definitions.
Linux	A UNIX-like Operating System whose name derives from the principal author of the Linux kernel, Linus Torvalds, and the "X" in UNIX.
Module	A generic word for DLL. See Dynamically Linked Library .
MMU	Memory Management Unit.
Overlay region	A region of memory which may overlap with one or more other regions of memory.
PLT	See Procedure Linkage Table .
PLTGOT	An inter-file procedure linkage using function addresses stored in a subsection of the global object table (GOT).
Procedure Linkage Table	A table of veneers for dynamic symbol references to procedures.
Relocation	An address or piece of information generated by a compiler or linker that must be resolved at a later build step.
RVCT	A suite of ARM tools including armcc, armasm, armlink, fromelf and C/C++ libraries.
Scatter-loading	The scatter-loading mechanism enables you to specify the memory map of an image to the linker using a description in a text file. Scatter-loading gives you complete control over the grouping and placement of image components.
Symbian	A group of open source operating systems and software platforms designed for smartphones.
Symdefs file	A file used to provide symbol versioning information to the linker.
Symbol Versioning	Extra information about a particular symbol imported from or exported from a shared library.
System V	One of the first commercial versions of the UNIX operating system.
SysV	See System V .
UNIX	An Operating System developed in 1969.
Windows	A series of Operating Systems developed by Microsoft.