C++ ABI for the ARM® Architecture

Document number: ARM IHI 0041D, current through ABI release 2.09

Date of Issue: 30th November 2012

Abstract

This document describes the C++ Application Binary Interface for the ARM architecture.

Keywords

C++ ABI, generic C++ ABI, exception handling ABI

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

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

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

Licence

Proprietary notice

Contents

1 ABOUT THIS DOCUMENT

1.1 Change control

1.1.1 Current status and anticipated changes

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

☐ Typographical corrections.

☐ Clarifications.

☐ Compatible extensions.

1.1.2 Change history

Issue	Date	By	Change
1.0	30th October 2003	LS	First public release.
2.0	24th March 2005	LS	Second public release.
2.01	4th July 2005	LS	Fixed defect in §3.2.4 – .init_array sections must be writable, but compiled *as if* read-only.
2.02	5th January 2006	LS	In §3.2.5.5, forbid the export of entities declared in unnamed namespaces.
2.03	3rd May 2006	LS	In §3.2.2.5, §3.2.4.2, and __aeabi_atexit, clarified the use of __aeabi_atexit().
2.04 / A	25th October 2007	LS	In §3.1, specified the name mangling (GC++ABI §5.1.5) for the 16-bit FP type added to [AAPCS] in ABI r2.06. Updated the base standard for C++ to ISO/IEC 14882:2003. Added an ARM-specific rule for the ELF binding of guard variable symbols (§3.2.6). Document renumbered (formerly GENC-003540 v2.04).
B	10th October 2008	LS	In §3.1, removed the ARM-specified mangling for 16-bit FP types added in r2.06 now that the [GC++ABI] defines it to be Dh; noted the mangling of std::va_list resulting from its definition in [AAPCS].
C	5th October 2009	LS	In §3.2.2.3, corrected typos in/ wording of the justification for defining __aeabi_vec_delete3 but not __aeabi_vec_delete2; in the definition of __aeabi_vec_ctor_nocookie_nodtor, corrected the order of *size* and *count* parameters to __aeabi_vec_ctor_cookie_nodtor(). In §3.2.5.5, corrected broken class export syntax; corrected comments about entities declared in unnamed namespaces and those derived from them.
D r2.09	30th November 2012	AC	In §3.1, clarified handling of empty classes.

1.2 References

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

Ref	URL or other reference	Title
AAPCS		Procedure Call Standard for the ARM Architecture
BSABI		ABI for the ARM Architecture (Base Standard)
CPPABI	*This document*	C++ ABI for the ARM Architecture
EHABI		Exception Handling ABI for the ARM Architecture
EHEGI		Exception handling components, example implementations
GC++ABI	http://mentorembedded.github.com/cxx-abi/abi.html	Itanium C++ ABI ($Revision: 1.71 $) (Although called *Itanium C++ ABI*, it is very generic).
Generic ELF	http://www.sco.com/developers/gabi/ …	Generic ELF, 17th December 2003 draft.
ISO C++	ISO/IEC 14882:2003 (14882:1988 with *Technical Corrigendum*)	International Standard ISO/IEC 14882:2003 – Programming languages C++

1.3 Terms and abbreviations

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

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

1.4   Your licence to use this specification

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

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

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

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

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

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

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

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

1.5 Acknowledgements

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

2   OVERVIEW

The C++ ABI for the ARM architecture (CPPABI) comprises four sub-components.

□   The generic C++ ABI, summarized in §2.1, is the referenced base standard for this component.

□   The *C++ ABI supplement* in §3.1 details ARM-specific additions to and deviations from the generic standard.

□   The separately documented *Exception Handling ABI for the ARM Architecture* [EHABI], summarized in §2.2, describes the language-independent and C++-specific aspects of exception handling.

□   The example implementations of the exception handling components [EHEGI], summarized in §2.3, include:

-    A language independent unwinder.

-    A C++ semantics module.

-    ARM-specific C++ unwinding personality routines.

The generic C++ ABI is implicitly an SVr4-based standard, and takes an SVr4 position on symbol visibility and vague linkage. The *C++ ABI supplement* in §3 details extensions for DLL-based environments.

2.1   The Generic C++ ABI

The generic C++ ABI [GC++ABI] (originally developed for SVr4 on Itanium) specifies:

□   The layout of C++ non-POD class types in terms of the layout of POD types (specified for *this* ABI by the *Procedure Call Standard for the ARM Architecture* [AAPCS]).

□   How class types requiring copy construction are passed as parameters and results.

□   The content of run-time type information (RTTI).

□   Necessary APIs for object construction and destruction.

□   How names with linkage are mangled (name mangling).

The generic C++ ABI refers to a separate Itanium-specific specification of exception handling. When the generic C++ ABI is used as a component of *this* ABI, corresponding reference must be made to the *Exception Handling ABI for the ARM Architecture* [EHABI] and §2.2.

2.2   The Exception handling ABI for the ARM architecture

In common with the Itanium exception handling ABI, the *Exception handling ABI for the ARM architecture* [EHABI] specifies table-based unwinding that separates language-independent unwinding from language specific aspects. The specification describes:

□   The *base class* format and meaning of the tables understood by the language-independent exception handling system, and their representation in relocatable files. The language-independent exception handler only uses fields from the base class.

□   A *derived table class* used by ARM tools that efficiently encodes stack-unwinding instructions and compactly represents the data needed for handling C++ exceptions.

□   The interface between the language independent exception handling system and the *personality routines* specific to a particular implementation for a particular language. Personality routines interpret the language-specific, derived class tables. Conceptually (though not literally, for reasons of implementation convenience and run-time efficiency), personality routines are member functions of the derived class.

☐ The interfaces between the (C++) language exception handling semantics module and:

    - The language-independent exception handling system.

    - The personality routines.

    - The (C++) application code (effectively the interface underlying *throw*).

The EHABI contains a significant amount of commentary to aid and support independent implementation of:

☐ Personality routines.

☐ The language-specific exception handling semantics module.

☐ Language-independent exception handling.

This commentary does not provide, and is not intended to provide, complete specifications of independent implementations, but it does give a rationale for the interfaces to, and among, these components.

2.3 The exception handling components example implementation

The exception handling components example implementation (EHEGI) comprises the following files.

☐ **cppsemantics.cpp** is a module that implements the semantics of C++ exception handling. It uses the language-independent unwinder (unwinder.c), and is used by the ARM-specific personality routines (unwind_pr.[ch]).

☐ **cxxabi.h** describes the generic C++ ABI (§2.1).

☐ **Licence.txt** describes your licence to use the exception handling example implementation.

☐ **unwind_env.h** is a header that describes the build and execution environments of the exception handling components. This header must be edited if the exception handling components are to be built with non-ARM compilers. This header #includes cxxabi.h.

☐ **unwind_pr.c** implements the three ARM-specific personality routines described in the *Exception Handling ABI for the ARM Architecture*.

☐ **unwinder.c** is an implementation of the language-independent unwinder.

☐ **unwinder.h** describes the interface to the language-independent unwinder, as described in the *Exception Handling ABI for the ARM Architecture*.

3 THE C++ ABI SUPPLEMENT

3.1 Summary of differences from and additions to the generic C++ ABI

This section summarizes the differences between the *C++ ABI for the ARM architecture* and the generic C++ ABI. Section numbers in captions refer to the generic C++ ABI specification. Larger differences are detailed in subsections of §3.2.

GC++ABI §1.2 *Limits*

The offset of a non-virtual base sub-object in the full object containing it must fit into a 24-bit signed integer (because of RTTI implementation). This implies a practical limit of 2^{23} bytes on the size of a class sub-object.

GC++ABI §2.2 POD Data Types

[AAPCS] describes the handling of C++ POD types which have an equivalent in Standard C. Consequently it does not describe the handling of C++ empty POD classes (Standard C forbids empty structures and unions). Such classes should be treated as if they contained a single char member containing an unspecified value.

GC++ABI §2.3 *Member Pointers*

The pointer to member function representation differs from that used by Itanium. See §3.2.1.

GC++ABI §2.7 *Array operator new cookies*

Array cookies, when present, are always 8 bytes long and contain both element size and element count (in that order). See §3.2.2.

GC++ABI §2.8 *Initialization guard variables*

Static initialization guard variables are 4 bytes long not 8, and there is a different protocol for using them which allows a guard variable to implement a semaphore when used as the target of ARM SWP or LDREX and STREX instructions. See §3.2.3.

GC++ABI §2.9.1 *Run-Time Type Information (RTTI), General*

The target platform ABI specifies whether address equality is required for type_info objects that describe the same type. (The ABI-defined symbol names for type_info objects and their names match the pattern _ZT{I,S}*). A C++ system that supports a platform must follow the platform's specification. The GC++ABI gives the correct specification for SVr4-based platforms such as Linux.

A C++ system must provide implementations of `std::type_info::operator==`, `std::type_info::operator!=`, and `(const std::type_info&)::before` appropriate to the target platform.

These std::type_info functions should not be inline by default, as doing so makes the relocatable file platform-specific. A C++ system must provide an option or default (Q-o-I) to force them out of line.

GC++ABI §3.1.5 *Constructor return values*

This ABI requires C1 and C2 constructors to return *this* (instead of being void functions) so that a C3 constructor can tail call the C1 constructor and the C1 constructor can tail call C2.

Similarly, we require D2 and D1 to return *this* so that D0 need not save and restore *this* and D1 can tail call D2 (if there are no virtual bases). D0 is still a void function.

We do not require thunks to virtual destructors to return *this*. Such a thunk would have to adjust the destructor's result, preventing it from tail calling the destructor, and nullifying any possible saving.

Consequently, only non-virtual calls of D1 and D2 destructors can be relied on to return *this*.

GC++ABI §3.3.2 *One-time construction API*

The type of parameters to __cxa_guard_acquire, __cxa_guard_release and __cxa_guard_abort is 'int*' (not '__int64_t*'), and use of fields in the guard variable differs. See §3.2.3.

GC++ABI §3.3.4 *Controlling Object Construction Order*

#pragma priority is not supported. See §3.2.4.1 for details of how global object construction is coordinated.

GC++ABI §3.3.5.3 *Runtime API*

This ABI defines __aeabi_atexit (§3.2.2.5 and __aeabi_atexit), for use in place of __cxa_atexit.

It is forbidden for user code to call __cxa_atexit or __aeabi_atexit directly, or for any call to __aeabi_atexit (other than ones from the implementations of the *atexit* library functions) to be executed more than once (§3.2.4.2).

GC++ABI §3.4 *Demangler API*

The demangler is not provided as a library.

GC++ABI §5.2.2 *Static Data (new in ABI r2.06)*

If a static datum and its guard variable are emitted in the same COMDAT group, the ELF binding [Generic ELF] for both symbols must be STB_GLOBAL, not STB_WEAK as specified in [GC++ABI]. §3.2.6 justifies this requirement.

GC++ABI §5.2.3 *Virtual Tables and the key function*

A compiler selects the key function for a class T when it has read the entire translation unit containing the definition of T. The key function is the textually first, non-inline, non-pure, virtual, member function of T.

An inline member is not a key function even if it is the first declared inline at the completion of the class definition.

(In contrast, the GC++ABI §5.2.3 defines the key function to be the textually first, non-inline, non-pure, virtual function identified at completion of the class definition).

In the following example, the key function is T::f.

```
struct T {
  inline virtual void a();      // inline
         virtual void b();      // might be defined inline later...
         virtual void c() { }   // implicitly inline
         virtual void d() = 0;  // pure
                 void e();      // not virtual...
         virtual void f(), g();
};
inline void T::b() { }          // but b is defined to be inline
// End of translation unit... The key function is 'T::f'; GC++ABI chooses T::b;
```

GC++ABI §5.3 *Unwind Table Location*

See §4 of *Exception Handling ABI for the ARM Architecture* [EHABI].

(No section in the generic C++ ABI – a library nothrow new function must not examine its 2nd argument)

Library versions of the following functions *must not* examine their second argument.

```
::operator new(std::size_t, const std::nothrow_t&)
::operator new[](std::size_t, const std::nothrow_t&)
```

(The second argument conveys no useful information other than through its presence or absence, which is manifest in the mangling of the name of the function. This ABI therefore allows code generators to use a potentially invalid second argument – for example, whatever value happens to be in R1 – at a point of call).

(No section in the generic C++ ABI – library placement new functions must be inline)

We require the library placement allocation functions (§18.4.1.3 of ISO C++) to be inline with these definitions:

```
inline void *operator new(std::size_t, void* __ptr) throw() { return __ptr; }
inline void *operator new[](std::size_t, void* __ptr) throw() { return __ptr; }
```

We do not require the library placement deallocation functions to be inline:

```
void operator delete(void*, void*) throw();
void operator delete[](void*, void*) throw();
```

(They can only be called via exceptions thrown by failing constructors or directly by user code).

(No section in the generic C++ ABI, but would be §2.2 POD data types)

Pointers to extern "C++" functions and pointers to extern "C" functions are interchangeable if the function types are otherwise identical.

In order to be used by the library helper functions described below, implementations of constructor and destructor functions (complete, sub-object, and allocating) must have a type compatible with:

```
extern "C" void* (*)(void* /* , other argument types if any */);
```

Deleting destructors must have a type compatible with:

```
extern "C" void (*)(void*);
```

(No section in the generic C++ ABI, but would be §3.3.4 Controlling Object Construction Order)

Global object construction and destruction are managed in a simplified way under this ABI (see §3.2.4).

(No section in the generic C++ ABI – DLL symbol visibility and linkage issues)

§3.2.5 discusses inter-DLL symbol visibility and linkage issues.

(No section in the generic C++ ABI – Namespace and mangling for the va_list type) **(new in r2.07)**

The type __va_list is in namespace std. The type name of va_list therefore mangles to St9__va_list.

3.2 Differences in detail

3.2.1 Representation of pointer to member function

The generic C++ ABI [GC++ABI] specifies that a pointer to member function is a pair of words *<ptr, adj>*. The least significant bit of *ptr* discriminates between (0) the address of a non-virtual member function and (1) the offset in the class's virtual table of the address of a virtual function.

This encoding cannot work for the ARM-Thumb instruction set where code addresses use all 32 bits of *ptr*.

This ABI specifies that *adj* contains twice the *this* adjustment, plus 1 if the member function is virtual. The least significant bit of *adj* then makes exactly the same discrimination as the least significant bit of *ptr* does for Itanium.

A pointer to member function is NULL when *ptr* = 0 *and* the least significant bit of *adj* is zero.

3.2.2 Array construction and destruction

3.2.2.1 Array cookies

An array cookie is used for heap-allocated arrays of objects with class type where the class has a destructor or the class's *usual (array) deallocation function* [ISO C++ §3.7.3.2] has two arguments, i.e. `T::operator delete(void*, std::size_t)`. Nonetheless, an array cookie is not used if `::operator new[](std::size_t, void*)` is used for the allocation as the user is then responsible for the deallocation and the associated bookkeeping.

When a cookie is needed this ABI always specifies the same cookie type:

```
struct array_cookie {
    std::size_t element_size; // element_size != 0
    std::size_t element_count;
};
```

This is different than the generic C++ ABI which uses a variable sized cookie depending on the alignment of element type of the array being allocated.

Note   Although it's not a particularly useful property, this cookie is usable as a generic C++ cookie when the generic C++ cookie size is 8 bytes.

Both the element size and element count are recorded in the cookie. For example, in the following the element size would be sizeof(S) = 8 and the element count would be 3 * 5 = 15.

```
struct S { int a[2]; };
typedef SA S[3];
S* s = new SA[5];
```

Note   The element size can never legally be zero. Finding a zero element size at delete [ ] time indicates heap corruption.

3.2.2.2 Array cookie alignment

The array cookie is allocated at an 8-byte aligned address immediately preceding the user's array. Since the cookie size is 8 bytes the user's array is also 8-byte aligned.

3.2.2.3 Library helper functions

The generic C++ ABI contains some helper functions for array construction and destruction:

```
__cxa_vec_new          __cxa_vec_new2
__cxa_vec_new3         __cxa_vec_ctor
__cxa_vec_dtor         __cxa_vec_cleanup
__cxa_vec_delete       __cxa_vec_delete2
__cxa_vec_delete3      __cxa_vec_cctor
```

Compilers are not required to use these helper functions but runtime libraries must supply them and they must work with the always 8-byte cookies. These functions take pointers to constructors or destructors. Since constructors and destructors conforming to this ABI return *this* (§3.1, ¶§3.1.5 *Constructor return values*, above) the return types of these parameters are void* instead of void.

The generic C++ ABI gives __cxa_vec_ctor and __cxa_vec_cctor a void return type. This ABI specifies void* instead. The value returned is the same as the first parameter – a pointer to the array being constructed. We do not change the return type for __cxa_vec_dtor because we provide __aeabi_vec_dtor which has the additional advantage of not taking a padding_size parameter.

In addition, we define the following new helpers which can be called more efficiently.

```
__aeabi_vec_ctor_nocookie_nodtor
__aeabi_vec_ctor_cookie_nodtor
__aeabi_vec_cctor_nocookie_nodtor
__aeabi_vec_new_cookie_noctor
__aeabi_vec_new_nocookie
__aeabi_vec_new_cookie_nodtor
__aeabi_vec_new_cookie
__aeabi_vec_dtor
__aeabi_vec_dtor_cookie
__aeabi_vec_delete
__aeabi_vec_delete3
__aeabi_vec_delete3_nodtor
__aeabi_atexit
```

Again, compilers are not required to use these functions but runtime libraries must supply them.

__aeabi_vec_dtor effectively makes __cxa_vec_dtor obsolete.

Compilers are encouraged to use the __aeabi_vec_dtor instead of __cxa_vec_dtor and __aeabi_vec_delete instead of __cxa_vec_delete. Run-time environments are encouraged to expect this, perhaps implementing __cxa_vec_delete in terms of __aeabi_vec_delete instead of the other way around.

We define __aeabi_vec_delete3 but not a corresponding __aeabi_vec_delete2. Using that would be less efficient than using __aeabi_vec_dtor and calling the `T1::operator delete[]` directly. See note 3 on page 18, below.

__cxa_vec_ctor still has uses not covered by __aeabi_vec_ctor_nocookie_nodtor and __aeabi_vec_ctor_cookie_nodtor.

Additional helpers for array construction (i.e. new T[n], __aeabi_vec_new_*) may be added in future releases of this ABI.

Definitions of the __aeabi_* functions are given below in terms of example implementations. It is not required to implement them this way.

```
#include <cstddef>  // for ::std::size_t
#include <cxxabi.h> // for __cxa_*

namespace __aeabiv1 {
  using ::std::size_t;

  // Note: Only the __aeabi_* names are exported.
  // array_cookie, cookie_size, cookie_of, etc. are presented for exposition only.
  // They are not expected to be available to users, but implementers may find them useful.

  struct array_cookie {
      size_t element_size; // element_size != 0
      size_t element_count;
  };

  // The struct array_cookie fields and the arguments element_size and element_count
  // are ordered for convenient use of LDRD/STRD on architecture 5TE and above.

  const size_t cookie_size = sizeof(array_cookie);

  // cookie_of() takes a pointer to the user array and returns a reference to the cookie.

  inline array_cookie& cookie_of(void* user_array)
  {
    return reinterpret_cast<array_cookie*>(user_array)[-1];
  }

  // element_size_of() takes a pointer to the user array and returns a reference to the
  // element_size field of the cookie.

  inline size_t& element_size_of(void* user_array)
  {
    return cookie_of(user_array).element_size;
  }

  // element_count_of() takes a pointer to the user array and returns a reference to the
  // element_count field of the cookie.

  inline size_t& element_count_of(void* user_array)
  {
    return cookie_of(user_array).element_count;
  }

  // user_array_of() takes a pointer to the cookie and returns a pointer to the user array.

  inline void* user_array_of(array_cookie* cookie_address)
  {
    return cookie_address + 1;
  }

  extern "C" void* __aeabi_vec_ctor_nocookie_nodtor(
          void* user_array,
          void* (*constructor)(void*),
          size_t element_size, size_t  element_count)
  { // The meaning of this function is given by the following model implementation...
    // Note: AEABI mandates that __cxa_vec_ctor return its first argument
    return __cxa_vec_ctor(user_array, element_count, element_size, constructor, NULL);
  }
```

```
// __aeabi_vec_ctor_cookie_nodtor is like __aeabi_vec_ctor_nocookie_nodtor but sets
// cookie fields and returns user_array. The parameters are arranged to make STRD
// usable.  Does nothing and returns NULL if cookie is NULL.

extern "C" void* __aeabi_vec_ctor_cookie_nodtor(
        array_cookie* cookie,
        void*(*constructor)(void*),
        size_t element_size, size_t element_count)
{ // The meaning of this function is given by the following model implementation...
  if (cookie == NULL){ return NULL; }
  else
  {
    cookie->element_size = element_size;  cookie->element_count = element_count;
    return __aeabi_vec_ctor_nocookie_nodtor(
            user_array_of(cookie), constructor, element_size, element_count);
  }
}

extern "C" void* __aeabi_vec_cctor_nocookie_nodtor(
        void* user_array_dest,
        void* user_array_src,
        size_t element_size, size_t element_count,
        void* (*copy_constructor)(void*, void*))
{ // The meaning of this function is given by the following model implementation...
  // Note: AEABI mandates that __cxa_vec_cctor return its first argument
  return __cxa_vec_cctor(user_array_dest, user_array_src,
          element_count, element_size, copy_constructor, NULL);
}

extern "C" void* __aeabi_vec_new_cookie_noctor(size_t element_size, size_t element_count)
{ // The meaning of this function is given by the following model implementation...
  array_cookie* cookie =
      reinterpret_cast<array_cookie*>
          (::operator new[](element_count * element_size + cookie_size));
  cookie->element_size = element_size; cookie->element_count = element_count;
  return user_array_of(cookie);
}

extern "C" void* __aeabi_vec_new_nocookie(
        size_t  element_size, size_t  element_count,
        void* (*constructor)(void*))
{ // The meaning of this function is given by the following model implementation...
  return __cxa_vec_new(element_count, element_size, 0, constructor, NULL);
}

extern "C" void* __aeabi_vec_new_cookie_nodtor(
        size_t  element_size, size_t  element_count,
        void* (*constructor)(void*))
{ // The meaning of this function is given by the following model implementation...
  return __cxa_vec_new(element_count, element_size, cookie_size, constructor, NULL);
}

extern "C" void* __aeabi_vec_new_cookie(
        size_t  element_size, size_t  element_count,
        void* (*constructor)(void*),
        void* (*destructor)(void*))
{ // The meaning of this function is given by the following model implementation...
  return __cxa_vec_new(element_count, element_size, cookie_size, constructor, destructor);
}
```

```
// __aeabi_vec_dtor is like __cxa_vec_dtor but has its parameters reordered and returns
// a pointer to the cookie (assuming user_array has one).
// Unlike __cxa_vec_dtor, destructor must not be NULL.
// user_array must not be NULL.

extern "C" void* __aeabi_vec_dtor(
        void* user_array,
        void* (*destructor)(void*),
        size_t element_size, size_t element_count)
{ // The meaning of this function is given by the following model implementation...
  __cxa_vec_dtor(user_array, element_count, element_size, destructor);
  return &cookie_of(user_array);
}

// __aeabi_vec_dtor_cookie is only used on arrays that have cookies.
// __aeabi_vec_dtor is like __cxa_vec_dtor but returns a pointer to the cookie.
// That is, it takes a pointer to the user array, calls the given destructor on
// each element (from highest index down to zero) and returns a pointer to the cookie.
// Does nothing and returns NULL if cookie is NULL.
// Unlike __cxa_vec_dtor, destructor must not be NULL.
//  Exceptions are handled as in __cxa_vec_dtor.
// __aeabi_vec_dtor_cookie must not change the element count in the cookie.
// (But it may corrupt the element size if desired.)

extern "C" void* __aeabi_vec_dtor_cookie(void* user_array, void* (*destructor)(void*))
{ // The meaning of this function is given by the following model implementation...
  // like:
  //   __cxa_vec_dtor(user_array, element_count_of(user_array),
  //                  element_size_of(user_array), destructor);
  return user_array == NULL ? NULL :
         __aeabi_vec_dtor(user_array, destructor,
                          element_size_of(user_array), element_count_of(user_array));
}

extern "C" void __aeabi_vec_delete(void* user_array, void* (*destructor)(void*))
{ // The meaning of this function is given by the following model implementation...
  // like:  __cxa_vec_delete(user_array, element_size_of(user_array),
  //                  cookie_size, destructor);
  try {
    ::operator delete[](__aeabi_vec_dtor_cookie(user_array, destructor));
  } catch (...) {
    if (user_array != NULL) {
      ::operator delete[](&cookie_of(user_array));
    }
    throw;
  }
}
```

```
extern "C" void __aeabi_vec_delete3(
        void* user_array, void (*destructor)(void*), void (*dealloc)(void*, size_t))
{ // The meaning of this function is given by the following model implementation...
  // like:  __cxa_vec_delete3(user_array, element_size_of(user_array),
  //                          cookie_size, destructor, decalloc);
  if (user_array != NULL) {
     size_t size =
         element_size_of(user_array) * element_count_of(user_array) + cookie_size;
    void *array_cookie;
    try {
      array_cookie = __aeabi_vec_dtor_cookie(user_array, destructor);
    } catch (...) {
      try {
        (*dealloc)(&cookie_of(user_array), size);
      } catch (...) {
        std::terminate();
      }
      throw;
    }
    (*dealloc)(array_cookie, size);
  }
}

extern "C" void __aeabi_vec_delete3_nodtor(
        void* user_array, void (*dealloc)(void*, size_t))
{ // The meaning of this function is given by the following model implementation...
  // like:  __cxa_vec_delete3(user_array, element_size_of(user_array),
  //                          cookie_size, 0, decalloc);
  if (user_array != NULL) {
     size_t size =
         element_size_of(user_array) * element_count_of(user_array) + cookie_size;
    (*dealloc)(&cookie_of(user_array), size);
  }
}

extern "C" int  __aeabi_atexit(void* object, void (*destroyer)(void*), void* dso_handle)
{ // atexit(f) should call __aeabi_atexit (NULL, f, NULL)
  // The meaning of this function is given by the following model implementation...
  return __cxa_atexit(destroyer, object, dso_handle);          // 0 ⇒ OK; non-0 ⇒ failed
}

} // namespace __aeabiv1
```

3.2.2.4 Code examples for the delete expression

Section 5.3.5 of the ISO C++ standard discusses the delete expression.

The code needed to implement **delete [] p** is tabulated in Table 1, below. It depends on:

☐   The static element type of p (referred to as T below),

☐   Which `operator delete []` is being used for this de-allocation: either `::operator delete(void*)` or `T1::operator delete(void*)`, where T1 is T or a base class of T.

☐   Whether a cookie is needed for arrays of T (see §3.2.2.1).

☐   *Has dtor*, which means T is a class type with a non-trivial destructor [ISO C++ §12.4]. In cases where there is no cookie there must be no dtor.

Table 1, Implementation of delete [] p

`operator delete []`	Needs cookie	*Has dtor*	Implementation of `delete [] p` / `::delete [] p`	Note
::operator delete[](void*)	N	–	::operator delete[](p)	
::operator delete[](void*)	Y	N	::operator delete[](&cookie_of(p))	2
		Y	__aeabi_vec_delete(p, &T::~T{D1})	
T1::operator delete[] (void*)	Y	N	T1::operator delete[](&cookie_of(p))	
		Y	T1::operator delete[] (__aeabi_vec_dtor_cookie(p, &T::~T{D1}))	
T1::operator delete[] (void*, std::size_t)	Y	N	__aeabi_vec_delete3_nodtor(p, &T1::operator delete[])	3
		Y	__aeabi_vec_delete3(p, &T::~T{D1}, &T1::operator delete[])	4

Notes

1. Other operator delete[]s, such as. operator delete[](void*, const std::nothrow&) or operator delete[](void*, void*), can be called explicitly by a user, but can only be called implicitly when a new array expression throws an exception during allocation or construction.

2. This is an unusual case that can only be reached by using `::delete[]`, for example:

   ```
   struct T { static void operator delete(void*, std::size_t); } *p;
   ::delete[] p;
   ```

3. `__aeabi_vec_delete3_nodtor(p, &T1::operator delete[])` could also be done this way:

   ```
    T1::operator delete[](&cookie_of(p), sizeof(T)*element_count_of(p))
   ```

4. `__aeabi_vec_delete3(p, &T::~T{D1}, &T1::operator delete[])` could also be done this way:

   ```
   T1::operator delete[]
       (__aeabi_vec_dtor_cookie(p, &T::~T{D1}), sizeof(T)*element_count_of(p))
   ```

3.2.2.5 Code example for __aeabi_atexit

Because constructors conforming to this ABI return *this*, construction of a top-level static object and the registration of its destructor can be done as:

```
__aeabi_atexit(T::T{C1}(&t), &T::~T{D1}, &__dso_handle);
```

This saves an instruction compared with calling __cxa_atexit directly, and allows a smart linker to calculate how much space to allocate statically to registering top-level object destructions (§3.2.4.2).

3.2.3 Guard variables and the one-time construction API

3.2.3.1 Guard variables

To support the potential use of initialization guard variables as semaphores that are the target of ARM SWP and LDREX/STREX synchronizing instructions we define a static initialization guard variable to be a 4-byte aligned, 4-byte word with the following inline access protocol.

```
            #define INITIALIZED 1

            // inline guard test…
            if ((obj_guard & INITIALIZED)!= INITIALIZED) {
                // TST obj_guard, #1; BNE already_initialized
                if (__cxa_guard_acquire(&obj_guard)) {
                    ...
            }
```

Usually, a guard variable should be allocated in the same data section as the object whose construction it guards.

3.2.3.2 One-time construction API

```
            extern "C" int __cxa_guard_acquire(int *guard_object);
```

If the guarded object has not yet been initialized, this function returns 1. Otherwise it returns 0.

If it returns 1, a semaphore might have been claimed and associated with *guard_object*, and either __cxa_guard_release or __cxa_guard_abort must be called with the same argument to release the semaphore.

```
            extern "C" void __cxa_guard_release(int *guard_object);
```

This function is called on completing the initialization of the guarded object. It sets the least significant bit of *guard_object* (allowing subsequent inline checks to succeed) and releases any semaphore associated with it.

```
            extern "C" void __cxa_guard_abort(int *guard_object);
```

This function is called if any part of the initialization of the guarded object terminates by throwing an exception. It releases any semaphore associated with *guard_object*.

3.2.4 Static object construction and destruction

3.2.4.1 Top-level static object construction

The compiler is responsible for sequencing the construction of top-level static objects defined in a translation unit in accordance with the requirements of the C++ standard. The run-time environment (helper-function library) sequences the initialization of one translation unit after another. The global *constructor vector* provides the interface between these agents as follows.

☐   Each translation unit provides a fragment of the constructor vector in an ELF section called .init_array of type SHT_INIT_ARRAY (=0xE) and section flags SHF_ALLOC + SHF_WRITE.

☐   Each element of the vector contains the address of a function of type extern "C" void (* const)(void) that, when called, performs part or all of the global object construction for the translation unit. Producers must treat .init_array sections *as if* they were read-only.

   The appropriate entry for an element referring to, say, __sti_*file* that constructs the global static objects in *file*cpp, is 0 relocated by R_ARM_ TARGET1(__sti_*file*). Usually, R_ARM_ TARGET1 is interpreted by a static linker as R_ARM_ABS32 (for details, see the [Note] below).

☐   Run-time support code iterates through the global constructor vector in increasing address order calling each identified initialization function in order. This ABI does not specify a way to control the order in which translation units are initialized.

Note   In some execution environments, constructor vector entries contain self-relative references, which cost an additional ADD in the library code that traverses the vector, but save dynamic relocations, giving a smaller executable size and faster start-up when an executable must be dynamically linked and relocated. In these environments, a static linker interprets R_ARM_TARGET1 as R_ARM_REL32 rather than as R_ARM_ABS32. In some execution environments, constructor vector entries will be allocated to a read-only execution segment.

3.2.4.2 Static object destruction

The sequencing of static object destruction in C++ requires destructions to be registered dynamically in the order of object construction (§3.2.2.5), correctly interleaved with any calls to the *atexit* library function(s).

This ABI requires static object destruction to be registered by calling __aeabi_atexit (§3.2.2.5 and [__aeabi_atexit]).

Implementations of the generic C++ ABI helper function __cxa_atexit usually allocate elements of the list of static objects to be destroyed dynamically, but some execution environments require static allocation. To support allocating this list statically, compilers must ensure that:

☐   Static object destructions are registered using __aeabi_atexit, *not* __cxa_atexit.

☐   Each call to __aeabi_atexit registers the destruction of the data objects constructed by the calling code. (Thus each static call will be executed at most once, and table-driven registration of several destructions by a single static call to __aeabi_atexit is forbidden).

The maximum number of destructions that can be registered by a relocatable file is then the number of sites calling __aeabi_atexit. A smart linker can count the number of sites and allocate space for the list accordingly.

The maximum number of calls to __aeabi_atexit on behalf of the *atexit* library functions is bounded by the implementation definition. The C++ standard requires at least 32 calls to be supported.

It is Q-o-I whether a linker and matching run-time library can allocate the list statically. So is the behaviour if the library calls __aeabi_atexit (e.g. on behalf of *atexit*) more times than a static allocation supports.

3.2.5 Inter-DLL symbol visibility and linkage

Strictly, only subsection 3.2.5.5 of this section contributes to this ABI. Subsections 3.2.5.1, 3.2.5.2, 3.2.5.3, 3.2.5.4, and 3.2.5.6 give background, terminology, and rationale, but do not form part of this standard.

3.2.5.1 Background

An SVr4 (Unix or Linux) dynamic shared object (DSO) is best thought of as a library, rather than a module with a controlled interface. By default, every global symbol defined in a DSO is visible to its clients. When a program is linked dynamically with a DSO, the linkage semantics are the same as when it is linked statically with the corresponding static library. (We ignore here DSOs loaded dynamically by dlopen(), for which there is no static counterpart). In this environment the C++ ABI need not be aware of the existence of DSOs, and, indeed, the generic C++ ABI hardly mentions them.

In contrast, a dynamic link library (DLL) is much more a module with a controlled interface. Historically, the visibility of symbols between DLLs has been controlled explicitly using import and export directives to the static linker or source code annotations such as the __declspec(dllexport) and __declspec(dllimport) familiar to Microsoft Windows developers. By default, global symbols defined in a DLL are invisible outside of it.

In C, there is a one to one correspondence between source entities with external linkage and global symbols. There are no implicit global symbols other than compiler helper functions. It is, therefore, tractable to control visibility explicitly (using a variety of Q-o-I mechanisms).

In C++ there are several implicit entities associated with classes (v-tables, RTTI, etc) that have corresponding C++ ABI-specified global symbols, but there is no simple, universally accepted model of controlling their visibility between DLLs. This ABI specifies a simple binary interface that promotes inter-operation between independently compiled relocatable files while remaining faithful to the DLL-based expectation of explicit visibility control.

A further complication is that, at the time of writing, not all DLL-based execution environments encompassed by the *ABI for the ARM Architecture* are capable of resolving vague linkage (§3.2.5.4) dynamically. This means that they cannot always provide a single address for entities required to have a single definition (§3.2.5.6).

3.2.5.2 Symbol visibility, DLL export, and DLL import

At the binary interface, the scope of an ELF global symbol is restricted by giving it a non default visibility attribute. Specifically, STV_HIDDEN restricts the visibility of a symbol to the executable file that defines it.

In effect, STV_DEFAULT implements DLL export and STV_HIDDEN implements DLL no export.

The source annotation denoting export is Q-o-I, but we expect `__declspecl([no]dllexport)` to be widely used.

Exporting a function that can be inlined should force the creation and export of an out-of-line copy of it. (See also [*Importing*]…, below).

When compiling for an SVr4-based environment, symbols with global binding should have default visibility by default (unless source annotation or tool options dictate otherwise). The C++ ABI does not change this.

When compiling for a DLL-based environment, we start from the position that symbols with global binding should have STV_HIDDEN visibility by default (unless source annotation or tool options dictate otherwise). This C++ ABI modifies this starting point as described in §3.2.5.5.

In some DLL models, addressing an imported datum requires an additional level of indirection compared with addressing a locally defined one. In these models, DLL import makes a compile-time distinction between a reference to a datum exported by some other DLL and one defined by this DLL.

(**Aside**: Under the SVr4 DSO model, all global symbols are addressed indirectly, whether imported or not, so no source annotation is needed. This supports pre-emption of any DSO definition at dynamic link time and allows vague linkage to be implemented dynamically. **End aside**).

The source annotation denoting import is Q-o-I, but we expect `__declspecl([no]dllimport)` to be widely used.

It is Q-o-I whether importing a definition also exports it, whether exporting a reference also imports it, and how these annotations interact with compiler steering options. Nevertheless, in ELF, restricting the visibility of an undefined symbol restricts the visibility of a definition that matches it at static link time.

Importing a function that can be inlined should suppress the creation of an out-of-line copy of it, the imported reference being used instead. (See also [*Exporting*]…, above).

3.2.5.3 Symbol visibility for C++ entities

Many C++ entities with linkage map one to one via C++ ABI-defined name mangling [GC++ABI] to corresponding ELF symbols with global binding. Examples include many data objects and class member functions. In principle, the export of these entities can be controlled explicitly by source annotation, just as in C.

Some C++ ABI-defined global symbols are associated uniquely with an entity of the above sort. Examples include static data local to a function that might be inlined and the initialization guard variables associated with it. In these cases, symbol visibility must follow that of the export controlled C++ entity (here, the function itself).

Remaining C++ ABI-defined global symbols relate to class impedimenta – virtual tables and RTTI. Under the generic C++ ABI they are the global symbols _ZT{V,T,I,S}*type*, where *type* is a mangled class name.

3.2.5.4 Vague linkage

Some C++ entities (including class impedimenta, out of line copies of inline functions, and the static data and string literals belonging to them) can have *vague linkage.*

Entities with vague linkage are defined in many relocatable files linked to form an executable file. Duplication is avoided using COMDAT groups [GC++ABI], so there is at most one definition in a DLL, DSO, or executable file.

To ensure a single definition program wide requires pre-emption of all but one definition at dynamic link time. In turn this requires that references to a DLL-local definition can be relocated at dynamic link time.

Class impedimenta and some other class entities have vague linkage unless the class has a *key function*. The translation unit containing the definition of the key function provides a unique point of definition for the impedimenta. Otherwise, definitions must be emitted wherever they are used.

3.2.5.5 Inter-DLL visibility rules for C++ ABI-defined symbols

For terminology, please refer to §3.2.5.2, §3.2.5.3, and §3.2.5.4.

Among C++ entities with linkage, only classes are exported by default (in the absence of Q-o-I source annotations and compiler options). No ELF symbol directly represents a class.

If a C++ ABI-defined global (CAG) symbol Y names an entity associated with a C++ function or data object X:

☐ Y must be exported if, and only if, X is exported.

☐ Y must be addressed as imported if X is addressed as imported.

If a CAG symbol Y names one of the impedimenta associated with an exported class X:

☐ If X has a [key function] K:

   - Y is exported from the DLL containing the translation unit that defines K.

   - Y is addressed as imported in every other DLL containing a translation unit that refers to X and uses Y.

☐ Otherwise, if X has no [key function]:

   - Y is both exported from, and addressed as imported in, each DLL that refers to X and uses Y[1].

Strictly, as far as this ABI is concerned, the control of export is Q-o-I. However, to foster inter-operation between compilers we require that:

☐ A class should be exported unless explicitly tagged otherwise (e.g. by class __delcspec(nodllexport) X…).

☐ A member of an implicitly exported class should be exported only if explicitly tagged as such (e.g. by __delcspec(dllexport) C::f(…) {…}).

☐ If a class is *explicitly* exported (e.g. by __delcspec(dllexport) class X …) and no class member is explicitly exported then all class members should be exported.

☐ These rules apply to each class individually. Explicitly exporting a class X does not implicitly export any base class of X, or any class derived from X.

Some names, despite formally having external linkage, are not usable outside the translation unit in which they are declared. Names to which this applies

☐ Are declared in unnamed namespaces.

☐ Have external C++ linkage but *not* `extern "C"` linkage.

Whether such names have local or global binding is Q-o-I but they must not have dynamic linkage.

3.2.5.6 One definition rule considerations in the absence of dynamic vague linkage

The last rule given in the first half of §3.2.5.5 ("[Otherwise], …") ensures that a DLL-based system capable of resolving vague linkage dynamically can give unique (within the program) addresses to the impedimenta associated with a class that has no key function.

As observed in §3.2.5.4, other C++ entities suffer from vague linkage, but, ultimately, these are all functions, or associated with functions. If a system cannot resolve vague linkage dynamically, a few simple rules that can be backed by compiler warnings will serve to alert programmers to most potential problems with multiple definitions.

---

[1] See §3.2.5.6 for a discussion of this rule and possible optimizations of it.

For example, a compiler might warn of the following in relation to a function with vague linkage.

☐ Taking its address (it will yield different results in different DLLs).

☐ Using function local static data (they will be different data in different DLLs).

☐ Taking the address of a string literal, or passing a string literal to other than a library function (the literal will have a different address in different DLLs, and this might matter if the address rather than the value is stored).

In short, it is feasible and reasonable for a system to avoid these problems through its programming standards.

However, usage of the class impedimenta cannot be regulated through programming conventions, so we need different rules for them. Specifically, we must drop the requirement that one definition should mean one address. This appears to have no consequence for virtual tables (symbols matching _ZT{V,T}*type*), as nothing seems to depend on the address of a virtual table being unique, but it matters for RTTI (symbols matching _ZT{I,S}*type*).

This runs contrary to §2.9.1 of [GC++ABI] which states:

☐ It is intended that two type_info pointers point to equivalent type descriptions if and only if the pointers are equal. An implementation must satisfy this constraint, e.g. by using symbol preemption, COMDAT sections, or other mechanisms.

Fortunately, we can ignore this requirement without violating the C++ standard provided that:

☐ type_info::operator== and type_info::operator!= compare the strings returned by type_info::name(), not just the pointers to the RTTI objects and their names.

☐ No reliance is placed on the address returned by type_info::name().
(That is, t1.name() != t2.name() does not imply that t1 != t2).

The first condition effectively requires that these operators (and type_info::before()) must be called out of line, and that the execution environment must provide appropriate implementations of them. A relocatable file built this way is oblivious to whether or not RTTI objects have unique addresses.

Finally we need to revisit the last rule of 3.2.5.5 ("[Otherwise], …"). It states:

☐ If X is exported but has no key function, Y is both exported from, and addressed as imported in, each DLL that refers to X and uses Y.

For any Y for which the execution environment waives the one address rule, these requirements are pointless. The exported Y will never be used, and because the system cannot resolve vague linkage dynamically, there is no need to address the entity named by Y as imported.

Import potentially affects code generation, so this involves a compile time decision. However, we can note that a system that imports indirectly – using an extra indirection to access imported data – can in principle resolve vague linkage dynamically, while one that does not in general cannot. (But note that SVr4 applications do precisely this by using copy relocations and a local copy of the data that pre-empts any DSO copy). So, in practice, code generation is unlikely to be changed unless a system capable of resolving vague linkage dynamically chooses not to do so for class impedimenta, or unless code is generated differently for applications and DLLs.

Export does not affect code generation directly – merely the visibility of symbols and, hence, the efficiency of export tables. So it is desirable to drop the export requirement in environments that waive the one address rule. Doing this at compile time restricts the portability of the relocatable file. However it is easy to do the restriction at static link time, as follows.

☐ For any global symbol Y whose name matches _ZT{V,T,S,I}*type*, if Y is defined in a section belonging to a COMDAT group, reduce Y's visibility to STV_HIDDEN.

(If Y names an entity whose linkage is not vague, it will not be defined in a COMDAT group).

Whether implemented at compile time or link time, support by tool chains is Q-o-I.

3.2.6 ELF binding of static data guard variable symbols

The generic C++ standard [GC++ABI] states at the end of §5.2.2:

> *Local static data objects generally have associated guard variables used to ensure that they are initialized only once (see 3.3.2). If the object is emitted using a COMDAT group, the guard variable must be too. It is suggested that it be emitted in the same COMDAT group as the associated data object, but it may be emitted in its own COMDAT group, identified by its name. In either case, it must be weak.*

In effect the generic standard permits a producer to generate one of two alternative structures. Either:

```
COMDAT Group (Variable Name) {
    Defines Variable Name            // ELF binding STB_GLOBAL, mangled name
    Defines Guard Variable Name      // ELF binding STB_WEAK, mangled name ...
}                                    // (... this ABI requires STB_GLOBAL binding)
```

Or:

```
COMDAT Group (Variable Name) {
    Defines Variable Name            // ELF binding STB_GLOBAL, mangled name
}
+
COMDAT Group (Guard Variable Name) {
    Defines Guard Variable Name      // ELF binding STB_WEAK, mangled name
}
```

A link step involving multiple groups of the first kind causes no difficulties. A linker must retain only one copy of the group and there will be one definition of *Variable Name* and one weak definition of *Guard Variable Name*.

A link step involving pairs of groups of the second kind also causes no difficulties. A linker must retain one copy of each group so there will be one definition of *Variable Name* and one weak definition of *Guard Variable Name*.

A link step involving a group of the first kind and a pair of groups of the second kind generates two sub-cases.

☐ If the linker discards the group that defines two symbols there is no problem.

☐ If the linker retains the group that defines both *Variable Name* and *Guard Variable Name* it must nonetheless retain the group called *Guard Variable Name*. There are now two definitions of *Guard Variable Name* with ELF binding STB_WEAK.

In this second case there is no problem provided the linker picks one of the definitions.

Unfortunately, [Generic ELF] does not specify how linkers must process multiple weak definitions when there is no non-weak definition to override them. If a linker faults duplicate weak definitions there will be a functional failure.

This ABI requires the ELF binding of *Guard Variable Name* in the first structure to be STB_GLOBAL.

The rules codified in [Generic ELF] then make all three linking scenarios well defined and it becomes possible to link the output of compilers such as armcc that choose the first structure with the output of those such as gcc that choose the second without relying on linker behavior that the generic ELF standard leaves unspecified.